Universidad Complutense de Madrid

Facultad de Informática

Proyecto de Sistemas Informáticos

IMPLEMENTACIÓN Y EVALUACIÓN DE UN REPERTORIO DE INSTRUCCIONES MULTIMEDIA

Profesor Director: Luis Piñuel Moreno

Autores: Jacobo Cadavid Zaragoza
Jon Crespo Anasagasti
José Manuel Díaz Ruiz

RESUMEN DEL PROYECTO

- Resumen:

  La importancia que han ido adquiriendo los programas multimedia y de comunicaciones en los últimos años tanto en el mercado de Pcs como en la electrónica de consumo en general, ha propiciado que los principales fabricantes de procesadores les hayan ido incorporado nuevas instrucciones de tipo SIMD (MMX, SSE, AltiVec) para acelerar este tipo de programas.

  El objetivo de nuestro proyecto es implementar un subconjunto de instrucciones AltiVec sobre un simulador de arquitectura PowerPC (IBM/Freescale). De esta manera se amplían las posibilidades de la oferta de simulación arquitectónica para esta tecnología.

- Summary:

  The importance which multimedia and broadcasting's programs have been acquiring in the last years as much in the Pc's bussiness as in the electronics of general consumption, has caused that the main processor's manufacturers have been incorporating new SIMD instructions to acceleratte this type of programs.

  The target of our project is to implement a subset of AltiVec instruction on a PowerPC architecture simulator. This way, the offer possibilities of architectonic simulation for this technology are extended.

PALABRAS CLAVE

AltiVec
PowerPC
Dynamic SimpleScalar
Procesador vectorial
Simulador arquitectónico
SIMD
Viterbi
Registro vectorial
Repertorio de instrucciones

ÍNDICE

1 INTRODUCCION

Un simulador arquitectónico es una herramienta que se usa para simular el funcionamiento de diferentes arquitecturas. Los simuladores han sido mejorados progresivamente y pueden ejecutar códigos que se aproximen a tamaños reales en cantidades tratables de tiempo. En un Pentium Pro a 200Mhz el simulador diseñado que ofrece menos detalles de la ejecución, el mas rápido, simula aproximadamente cuatro millones de ciclos máquina por segundo cuando el más detallado simula 150.000 por segundo.

La arquitectura que nosotros vamos a tratar es la arquitectura PowerPC, la cual provee un modelo software que asegura software compatible entre diferentes implementaciones de la familia de microprocesadores PowerPC. La arquitectura PowerPC es una arquitectura de 64 bits, con un subconjunto de 32 bits. Los registros enteros tendrán capacidad de 32 o 64 bits dependiendo de la arquitectura sobre la que trabajemos, mientras que en todos los casos los registros en punto flotante serán de 64 bits. En general, la arquitectura PowerPC se define de la siguiente manera:

- El conjunto de instrucciones está formado por conjuntos de familias específicas (lógicas, load, store, aritméticas, punto flotante), instrucciones específicas, y las formas de codificación de las instrucciones. También se especifican los direccionamientos para realizar los accesos a memoria.

- El modelo de programación define el conjunto de registros y las convenciones de memoria.

- El modelo de memoria define el tamaño del espacio de direcciones y de sus subdivisiones. También define la forma de configurar las páginas y bloques de memoria con respecto al ordenamiento de los bytes, la coherencia y los distintos tipos de protección de la memoria.

- El modelo de mantenimiento de la memoria define cómo se particiona la memoria, cómo se configura y cómo se protege.

La arquitectura PowerPC, desarrollada en conjunto por Motorota, IBM, y Apple Computer, se basa en la arquitectura POWER implementada para la familia de computadores RS/6000. La arquitectura PowerPC, toma ventaja de los recientes avances tecnológicos en áreas tales como tecnología de proceso y diseño de compilador y reduce el conjunto de instrucciones. La arquitectura PowerPC es una arquitectura flexible y escalable.

Las instrucciones vectoriales se utilizan para realizar la misma operación sobre un conjunto de datos a la vez, en vez de hacerlo de una en una. Es un recurso habitual para el procesamiento en paralelo. En concreto, el repertorio de instrucciones AltiVec se ejecutan sobre el procesador PowerPC. AltiVec es un conjunto de instrucciones SIMD en coma flotante y enteros diseñado y en propiedad de Apple Computer (**Velocity Engine**), IBM (**VMX**) y Motorola (**AltiVec**), y puesto en ejecución en las versiones de PowerPC incluyendo el G4 de Motorola y los procesadores G5 de IBM. El repertorio de instrucciones AltiVec soporta aplicaciones de audio y vídeo.

Nosotros nos vamos a centrar en la creación de los registros necesarios en el código para poder implementar el repertorio de instrucciones vectoriales, así como en las propias instrucciones. Dichos registros simularán los registros específicos de la arquitectura AltiVec para lo cual vamos a tener que crear un tipo nuevo de datos, así como varias macros de acceso a dichos registros. En cuanto a las instrucciones, las hemos ido implementando por distintos módulos, tales como instrucciones aritmético-lógicas, de carga, de almacenamiento, en punto flotante, etcétera, debido a las similitudes que existen entre las distintas instrucciones pertenecientes a cada grupo.

2 SIMULADOR ARQUITECTÓNICO

Es una herramienta que se usa para simular el funcionamiento de diferentes arquitecturas mediante software.

Un simulador arquitectónico muy extendido es el SimpleScalar por su flexibilidad, facilidad de uso y eficacia.

Esta herramienta nos permite simular la arquitectura PowerPC, aunque solamente en su implementación de 32 bits.

2.1 SimpleScalar

El paquete de herramientas SimpleScalar implementa una simulación rápida, flexible y precisa de los procesadores modernos. La herramienta toma binarios compilados para la arquitectura de SimpleScalar y simula su ejecución en varios simuladores dados. Usamos paquetes de binarios precompilados más una versión modificada de GNU GCC que permite compilar nuestros propios test de prueba en código C.

Las ventajas de las herramientas SimpleScalar son la alta flexibilidad, portabilidad y que es fácilmente ampliable. La herramienta es totalmente portable, haciendo uso únicamente de herramientas GNU. Este paquete de herramientas ha sido probado ampliamente en numerosas plataformas. Está creado para la fácil escritura de instrucciones sin tener que volver a fijar el objetivo del compilado para cambios posteriores. La definición de instrucciones hace fácil la escritura de nuevas simulaciones y hace que, las ya escritas, sean más fáciles de ampliar. Finalmente los simuladores han sido mejorados progresivamente y pueden ejecutar códigos que se aproximen a tamaños reales en cantidades tratables de tiempo. En un Pentium Pro a 200Mhz el simulador diseñado que ofrece menos detalles de la ejecución, el mas rápido, simula aproximadamente cuatro millones de ciclos maquina por segundo cuando el mas detallado simula 150.000 por segundo.

La versión actual de la herramienta ha sufrido una gran mejora desde la anterior versión. Incluye mayor documentación, compatibilidad con más plataformas, interfaces más claros, paquetes de manejo estadístico, un depurador y una herramienta para trazar la ejecución en el simulador fuera de orden.

La herramienta SimpleScalar simula la arquitectura PowerPC. Actualmente, sólo se contempla la implementación de 32 bits. Futuras versiones tal vez incluyan la arquitectura de 64 bits.

2.1.1 Descripción de la arquitectura

La arquitectura PowerPC tiene 224 instrucciones con 15 formatos de instrucción. No todas estas instrucciones están implementadas en el simulador. A continuación describiremos las características de la arquitectura que están implementadas en el simulador.

Los registros

La arquitectura PowerPC tiene definidos 32 registros de propósito general (GPR) y 32 registros de punto flotante (FPR), un registro de condición (CR), el registro de unión (LR) y el registro contador (CTR). El estado de la unidad de punto flotante se salva en un registro de 32 bits llamado registro de control de estado de punto flotante (FPSCR). Para las instrucciones en punto fijo existe un registro de 32 bits que contiene el estado de las excepciones generadas en su ejecución, llamado de Excepción en punto fijo (XER). Estos registros serán comentados en profundidad más adelante.

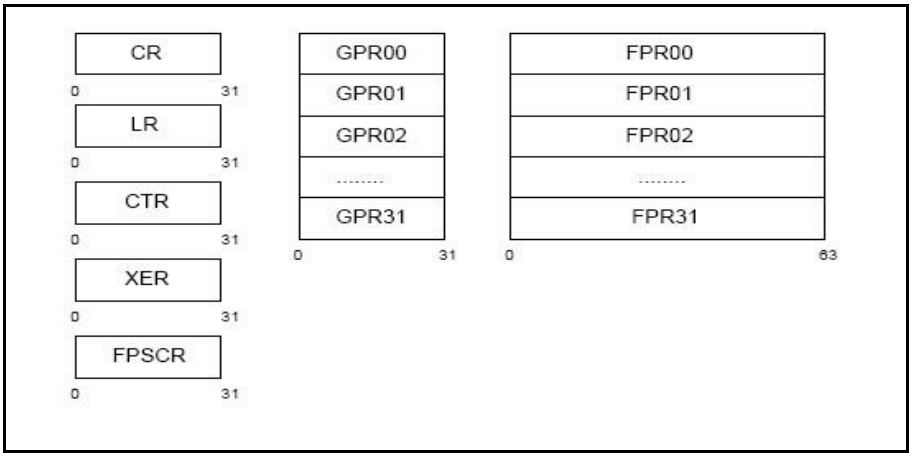

Figura 2.1 Registros [1]

Instrucciones

PowerPC cuenta con instrucciones de 4 bytes con alineamiento de palabra. Así, para una dirección de instrucción dada, los dos bits menos significativos son ignorados. Los bits 0-5 siempre dan el código de operación. Muchas de las instrucciones tienen un código de operación extendido. Algunas instrucciones tienen campos reservados que deben ser puestos a cero. Las instrucciones ilegales no definidas invocan al sistema de manejo de instrucciones ilegales. En el simulador, durante la decodificación de instrucciones, se invoca una llamada a *panic* que para la ejecución del simulador. No todas las instrucciones definidas en la arquitectura están implementadas en el simulador. Solo están definidas las instrucciones a nivel de usuario de 32 bits.

Modelo de almacenamiento

Los tipos de datos primitivos que define PowerPC son byte, media palabra y palabra. Los bytes se numeran en memoria de forma consecutiva empezando por el 0. Cada número es la dirección del byte correspondiente. Los operandos pueden ser de tamaño byte, media palabra, palabra o doble palabra. La dirección del operando es la dirección de su primer byte. Las direcciones sin alineamiento están permitidas para accesos a datos.

La arquitectura PowerPC soporta tanto *big-endian* como *little-endian* en el ordenamiento de byte, aunque en el simulador sólo está contemplado el formato *big-endian* (el bit más significativo en el bit 0).

2.1.2 Implementación

La herramienta SimpleScalar es modular y puede ser modificada para dar soporte a nuevas características de arquitecturas y microarquitecturas. Las diferentes estructuras simuladas como la caché, la memoria, los registros, la emulación de instrucciones y la microarquitectura están en diferentes archivos.

Todos los simuladores contenidos en la herramienta —sim-fast, sim-cache, sim-profile, sim-safe y sim-outorder— comparten estos archivos comunes.

2.1.3 Emulación de instrucciones

Los cinco simuladores comparten las definiciones de instrucciones contenidas en el archivo llamado *machine.def*. Este archivo contiene el código para la emulación de las instrucciones (en lenguaje C) así como el registro y dependencias funcionales de la instrucción. La corrección de las dependencias en una instrucción no afecta a su definición. Incluso si algunas dependencias estuvieran mal, el simulador funcional, el simulador de caché y la predicción de saltos funcionarían. Sin embargo, para el correcto funcionamiento del simulador de sincronización, estas dependencias deben estar bien definidas.

Algunas de las instrucciones están definidas sólo en el modo de 64 bits y el simulador se detiene con un error de instrucción ilegal cuando encuentra alguna de estas instrucciones.

La arquitectura PowerPC implementa la especificación aritmética de punto flotante según el estándar IEEE 754. El procesador de punto flotante eleva el número de excepciones y soporta cuatro modos de redondeo. Para simular el procesador de punto flotante se adopta una aproximación de dos puntos.

Dependiendo del host las instrucciones de punto flotante se ejecutan de forma nativa o son emuladas si el host es no-nativo.

Implementación nativa de punto flotante

La mayoría de las instrucciones de cómputo en punto flotante modifican un gran número de flags/campos en el FPSCR. Las instrucciones de cómputo son aquellas que realizan suma, resta, multiplicación, división, raíz cuadrada, redondeo, conversión, comparación, y combinaciones de estas operaciones.

En un host nativo, una emulación real del procesador en punto flotante se alcanza ejecutando la instrucción de forma nativa. Por emulación real entendemos que es la que realiza un cambio en la máquina simulada después de la ejecución igual que lo haría (en registros y memoria) en la máquina real.

Las variables de estado que se ven afectadas por una instrucción de cómputo en punto flotante son:

- Uno de los FPRs
- El registro de estado de punto flotante
- El registro de condición

El archivo de registro de la máquina simulada se salva como una variable en el simulador. El FPSCR y el CR son campos en este archivo de registro.

Para ejecutar una instrucción de cómputo en punto flotante se dan los siguientes pasos:

- Copiar el FPSCR de la máquina simulada en el FPSCR del host.
- Ejecutar la instrucción en punto flotante en el host en el que el simulador se está ejecutando. Esto afectará al estado del FPSCR en la máquina real. La salida generada por la ejecución se copia al archivo de registro del simulador.
- Copiar el valor del FPSCR desde el host en la estructura de datos del archivo de registro.

Implementación no nativa de punto flotante

En este caso las modificaciones sobre el FPSCR se ignoran. En un host no nativo el contenido del FPSCR se ignora. El modo de redondeo del compilador que se usa para compilar el simulador permanece siempre activo. En la simulación de los SPEC se aprecia que ignorar los cambios en el FPSCR no afecta a la ejecución.

Algunas de las instrucciones de cómputo en punto flotante modifican el registro de condición (CR). De acuerdo con el resultado de la instrucción, CR1 (los segundos cuatro bits de CR) se ponen a 0, 1, 2 ó 3. En el simulador se realiza comparando el resultado generado después de la ejecución. Este paso no varía entre los sistemas nativos y no nativos.

Accesos no alineados.

La arquitectura PowerPC permite las direcciones no alineadas para el acceso a datos. Para dar cabida a esto en el simulador, el alineamiento de cada lectura y escritura en memoria es comprobado y para cada no-alineamiento (lectura/escritura), las dos palabras consecutivas son leídas y los bytes correctos se juntan y se devuelven.

Toda lectura de palabra no alineada de memoria tiene como resultado dos lecturas simuladas y como consecuencia fallos de página y errores de caché simulados.

Cada escritura de palabra no alineada en memoria tiene como resultado dos lecturas de memoria para leer las dos palabras alineadas en los límites afectados por las lecturas, dos escrituras en memoria para escribir ambas palabras modificadas y consecuentemente los fallos de caché y fallos de página simulados que producen estos cuatro accesos.

```
#define FADD_IMPL                                  \
{                                                  \
      qword_T a,b;                                 \
      qword_T *dest;                               \
      double double_a,double_b,double_dest;\
      _a = PPCFPR_DW(RA);                          \
      _b = PPCFPR_DW(RB);                          \
      memcpy(&double_A,&_a,sizeof(double));\
      memcpy(&double_A,&_a,sizeof(double));\
      double_dest = double_a + double_b;    \
      dest = (qword_t *)(&double_dest);     \
      PPC_SET_FPR_DW(FD, *dest);            \
}
```

Figura 2.2 Implementación en un host no nativo

Hosts little-endian

El soporte para hosts *little-endian* está basado en las macros de memoria de SimpleScalar. En los hosts *little-endian* se trabaja reordenando los bytes antes de ser escritos o leídos de la memoria simulada. Durante la ejecución del programa a la memoria se accede de cuatro maneras:

- Cargando el programa: El cargador del sistema copia el segmento de código a memoria cuando se ha cargado el programa. En el simulador el código se lee del archivo binario y se escribe en la memoria simulada.
- Segmentos de datos, argumentos de programa y variables de entorno: Estos valores son escritos en memoria por el cargador.
- Llamadas al sistema: Algunas llamadas al sistema leen o escriben datos en buffers. Por ejemplo la llamada al sistema *fread* lee un bloque desde un archivo y lo escribe en un buffer en memoria.
- Instrucciones Load/Store: Instrucciones que leen o escriben valores de los registros en memoria.

Estos cuatro tipos de acceso a memoria pasan a través de la misma macro de acceso a memoria en el simulador. Para dotar de soporte para el paso entre *big-endian* y *little-endian*, en host *little-endian*, los bytes escritos en memoria son reordenados antes de escribirse y después de leerse de la memoria simulada. Reordenando los bytes de esta manera se garantiza que el contenido de la memoria sea *big-endian* independientemente de si el host lo es o no. Reordenando los contenidos usando las macros se obtienen los valores correctos en un host *little-endian* cuando éstos son usados en el cómputo de instrucciones de la sección de emulación del simulador. Las macros de acceso a memoria están definidas en *memory.h*.

2.1.4 Diferentes simuladores contenidos en SimpleScalar

SimpleScalar está compuesto por un conjunto de herramientas y de varios simuladores base. El código fuente está disponible de forma abierta, por lo que resulta fácilmente modificable para que los investigadores puedan adaptarlo a sus necesidades. SimpleScalar permite la simulación de múltiples Arquitecturas del Lenguaje Máquina; nosotros simulamos código PowerPC. Dentro del SimpleScalar podemos encontrar los siguientes simuladores implementados:

- *Sim-Outorder:* simulador temporal complejo de un procesador superescalar fuera de orden. De simulación más lenta, sim-outorder modela la temporización interna de un procesador ciclo a ciclo. Es el más complejo de todos y su ejecución es la más costosa en tiempo. Su tamaño ronda las 3900 líneas de código.

- *Sim-fast:* Es el simulador más rápido, no modela temporización ni genera ninguna salida especial, y además no realiza las comprobaciones de seguridad que si hace sim-safe. Ronda las 420 líneas de código.

- *Sim-profile:* Proporciona gran cantidad de información estadística. Muy adecuado para el modelado de aplicaciones sin necesidad de disponer del sistema.

- *Sim-cache sim-cheetah*: Estos simuladores son muy adecuados si el efecto de la caché en tiempo de ejecución no se necesita. Realizan una simulación funcional aportando información sobre las estadísticas de la cache.

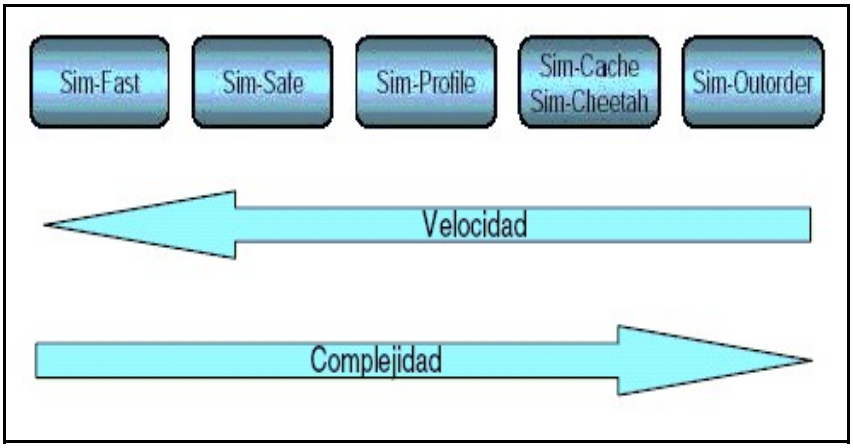

Figura 2.3 Simuladores [3]

2.1.4.1 Descripción de los archivos de código del simulador

- **bitmap.h:** Contiene macros para la manipulación de mapas de bits
- **bpred.[c,h]:** Lleva la creación, funcionalidad y actualización de predictores de salto.
- **cache.[c,h]:** Contiene funciones generales para soportar múltiples tipos de caché. Usa liked-lists para la comparación en cache de baja asociatividad y tablas hash para caches con alta asociatividad.
- **dlite.[c,h]:** Contiene el código del depurador Dlite.
- **endian.[c,h]:** define un grupo de funciones simples que determinan el orden de byte y de palabra en las plataformas host y destino
- **eval.[c,h]:** Contiene código para evaluar las expresiones usadas en Dlite.
- **eventq.[c,h]:** Define macros y funciones para mantener ordenadas las colas de eventos (usada para el ordenamiento del write back).
- **loader.[c,h]:** Carga el programa objeto en la memoria. Define el tamaño de segmento y las direcciones, define la primera pila de llamadas y obtiene el punto de comienzo del programa a ejecutar.
- **machine.[c,h]:** Contiene las llamadas a los registros y las instrucciones.
- **machine.def:** En este archivo están implementadas todas las instrucciones a nivel usuario.
- **main.c:** Realiza toda la inicialización y lanza la función principal del simulador.

- **memory.[c,h]:** Contiene las funciones para leer desde, escribir en, inicializar y borrar el contenido de la memoria principal. La memoria se implementa como un espacio dividido, en el que cada porción es adjudicada por petición.

- **misc.[c,h]:** Contiene numerosas funciones útiles como panic(), fatal(), debug().

- **options.[c,h]:** Contiene las opciones del paquete de código de SimpleScalar. Se usa para procesar los argumentos de la línea de comandos y las opciones especificadas en los archivos de configuración. Las opciones se registran en una base de datos.

- **ptrace.[c,h]:** Contiene código para producir y tomar las trazas del pipeline desde sim-outorder.

- **range.[c,h]:** Tiene el código que interpreta los comandos de rango de programa usados en Dlite.

- **regs.[c,h]:** Contiene funciones para inicializar los archivos de registro y vaciar su contenido.

- **resource.[c,h]:** Contiene código para manejar las unidades funcionales, divididas en clases. Las funciones definidas en el árbol crean tablas de ocupación y los grupos de recursos tomando un recurso de su grupo específico o si está disponible.

- **sim.[c,h]:** Contiene unas pocas declaraciones de variables externas y prototipos de funciones.

- **stats.[c,h]:** Contiene rutinas para manejar estadísticas midiendo así el comportamiento del programa ejecutado. Como con las opciones del paquete, los contadores son registrados por tipo con una base de datos interna. El paquete de estadísticas permite también medir distribuciones.

- **symbol.[c,h]:** Maneja los símbolos de programa y las líneas de información (se usa en Dlite).

- **syscall.[c,h]:** Contiene el código que funciona como interfaz entre las llamadas al sistema de SimpleScalar y las llamadas al sistema del host en el que se ejecute la herramienta.

- **sysprobe.c:** Comprueba el orden de byte y palabra de la plataforma del host y genera los flags adecuados.

- **version.h:** Define el número de versión y la fecha de revisión de la distribución.

2.2 Dynamic SimpleScalar (DSS)

El DSS es una herramienta que simula los programas de Java que funcionan en una JVM, usando la compilación *just in time*, ejecutando en una simulación multimodo sobre un procesador superescalar con ejecución fuera de orden con un sofisticado sistema de memoria. El DSS implementa soporte para la generación de código de forma dinámica, hilos, sincronización y mecanismos generales de señalización que soportan lanzamiento y recuperación de excepciones.

La microarquitectura modelada por SimpleScalar se distingue de la plataforma PowerPC, así que los resultados obtenidos difieren perceptiblemente y no es posible una validación por ciclo. Sin embargo el DSS alcanza los resultados que siguen de cerca la ejecución.

En SimpleScalar el código simulado se predecodifica antes del comienzo de la simulación, acelerando dicha simulación y emulando la instrucción de forma más eficiente. Esto se lleva a cabo mirando en la función qué código de operación de la instrucción simula, y sustituyendo las instrucciones en la memoria simulada con punteros a las funciones simuladas por esas instrucciones.

En los sistemas compilados dinámicamente es necesario generar y modificar el código durante la ejecución. Las dos posibilidades son redescifrar el código nuevo o modificado, o codificar la instrucción que se está ejecutando.

Otra modificación en el DSS es agregar un modelo de memoria virtual con ayuda para detectar violaciones de segmento.

En cuanto a los simuladores contenidos en el DSS son los mismos que en el SimpleScalar.

3 EXTENSIONES SIMD

Las arquitecturas SIMD se basan en el concepto de paralelismo de datos. La misma instrucción se aplica sobre un conjunto de operandos.

Es un campo muy atractivo para las aplicaciones multimedia tales como juegos, aplicaciones de audio y vídeo, es decir, programas que precisan muchos cálculos complejos. Por todo ello, estas arquitecturas se han ido ampliando y mejorando progresivamente en función de las necesidades que han ido apareciendo.

La tecnología AltiVec es una de estas extensiones que afianza la mejora del procesamiento vectorial de las instrucciones sobre PowerPC.

3.1 Introducción

SIMD es el acrónimo de *Single Instruction Multiple Data*, o Instrucción Única para Múltiples Datos. Los repertorios SIMD consisten en instrucciones que aplican una misma operación sobre un conjunto más o menos grande de datos. Es una arquitectura habitual para el procesamiento en paralelo con procesadores múltiples (que pueden llegar a centenares o miles) que realizan las mismas operaciones al mismo tiempo. Los resultados de estas computaciones en paralelo se reunifican para determinar un cierto resultado.

Todas estas unidades de procesamiento se encuentran bajo la supervisión de una única unidad de control común, quien despacha las instrucciones a diferentes unidades de procesamiento. Todos los procesadores reciben la misma instrucción de la unidad de control, pero operan sobre diferentes conjuntos de datos. Es decir, la misma instrucción es ejecutada de manera sincrónica por todas las unidades de procesamiento. Algunas máquinas comerciales con esta arquitectura son el Illiac IV, MPP y DAP.

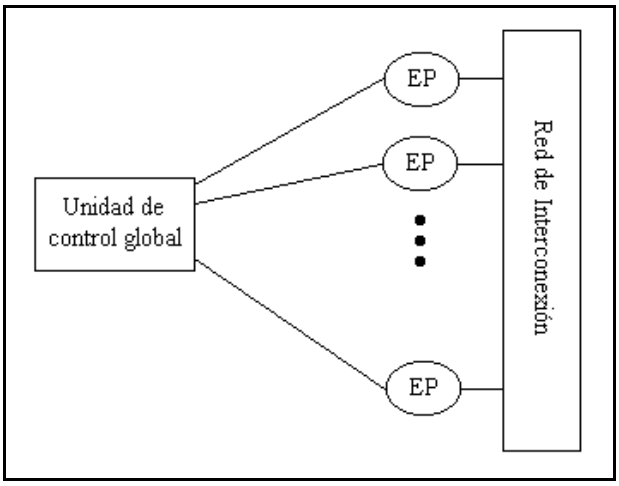

Figura 3.1 Procesador vectorial [13]

Los microprocesadores de propósito general modernos incluyen, desde hace tiempo, una serie de extensiones SIMD que les permiten trabajar con tipos de datos vectoriales. El uso de estas tecnologías permite obtener mejoras de rendimiento considerables, pero con un alto precio, pues se ha de programar en ensamblador y el código no es portable a otras arquitecturas, e incluso entre distintos modelos de la misma.

3.1.1 SIMD frente a procesadores vectoriales clásicos.

La vectorización ha sido una técnica destinada tradicionalmente a procesadores vectoriales. Las máquinas vectoriales se introdujeron en los años 70, con el fin de aumentar el uso del procesador reduciendo el tiempo de inicialización de los operadores y manteniendo llena constantemente la cadena de la segmentación. Para aprovechar estas ventajas, los programas que se creaban se escribían en términos de operaciones vectoriales sobre arrays completos en lugar de en forma de bucles operando componente a componente del vector.

La autovectorización de código escalar persigue que sea el propio compilador en la fase de optimización quien se encargue de transformar bucles de operaciones escalares en instrucciones vectoriales de la máquina de destino. De esta forma un bucle escalar se podría optimizar mediante instrucciones vectoriales, pero conservando la portabilidad del código original.

```
Código escalar                      Código vectorial
for (i=0; i<N; i++)                 a[0:N] = a [0:N] + b [0: N];
a[i] = a[i] + b[i];
```

Figura 3.2 Código escalar vs. vectorial

La teoría de la vectorización es un área de investigación madura. Desde los años 70 ha estado orientada a averiguar qué partes de un programa no tienen dependencias de datos entre ellas y pueden ser ejecutadas simultáneamente sin alterar el significado del programa original. Esta teoría clásica no es fácilmente aplicable a los procesadores SIMD.

Desde un punto de vista arquitectónico, los procesadores SIMD tienen diseños mucho más sencillos que los de los procesadores vectoriales tradicionales: sólo permiten accesos a posiciones contiguas de memoria, sin separación, y con restricciones de alineamiento. Los modos de direccionamiento son más simples y los registros son más pequeños. Aunque cuentan con operaciones para mitigar estas carencias, como las instrucciones de empaquetado, desempaquetado y reordenación, éstas son complejas y suponen un retraso que se añade al producido por la memoria.

Por si esto fuera poco, los juegos de instrucciones SIMD son de aplicación más restringida: no todas las operaciones están disponibles para todos los operandos, y las características de los juegos de instrucciones difieren de unas arquitecturas a otras. Por tanto, el trabajo del módulo de vectorización de un compilador no debe restringirse a un simple análisis de dependencias, sino que debe abordar transformaciones en el código para hacer buen uso de la potencia de las instrucciones de las que dispone el procesador.

En los últimos años han ido apareciendo procesadores con instrucciones SIMD que permiten realizar una misma operación sobre múltiples datos empaquetados dentro de un registro.

Estos registros son de un tamaño pequeño, y el número de elementos empaquetados que contienen es lo que determina el factor de paralelismo fp, que depende no sólo del tamaño del registro sino también del tamaño de los datos. Por ejemplo, en un registro de 128 bits, pueden caber 16 de 8 bits, 8 de 16 bits, 4 de 32, 2 de 64 o un entero de 128 bits.

Para realizar operaciones vectoriales se ha de separar el vector de fp en fp componentes e ir realizando las operaciones por separado. Esta operación se denomina **“seccionamiento”.**

3.1.2 Generalidades.

En una arquitectura SIMD el paralelismo se logra por múltiples unidades de proceso llamadas Elementos de Procesamiento (PEs), cada una de las cuales es capaz de ejecutar una operación especializada autónomamente.

La arquitectura de los Procesadores Array y de los Procesadores Vectoriales SIMD se caracteriza por el hecho de que la misma operación es realizada en un momento dado sobre un gran conjunto de datos en todos los PEs. Las computadoras SIMD están especialmente diseñadas para realizar cómputos vectoriales sobre matrices o arrays de datos.

Dado que un PE no constituye una unidad de proceso central completa (CPU) y que no es capaz de funcionar independientemente, el sistema es en este caso de procesamiento paralelo y no un sistema multiprocesador.

Pueden utilizarse indistintamente los términos Procesadores Array, Procesadores Paralelos, y computadoras SIMD.

3.1.3 Procesador Array.

Un array sincrónico de procesadores paralelos se denomina un Procesador Array, el cual consiste de múltiples Elementos de Procesamiento (PEs) bajo la supervisión de una única unidad de control (CU).

Un procesador Array puede manejar un único flujo de instrucción y múltiples flujos de datos. En tal sentido, los procesadores Array son también conocidos como máquinas SIMD.

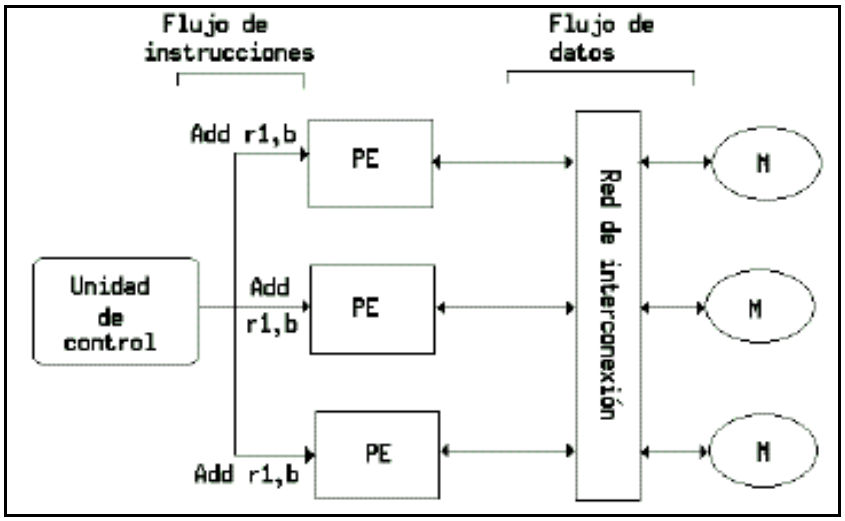

Figura 3.3 Ejecución SIMD [16]

3.2. AltiVec

La tecnología AltiVec se basa en la ejecución de instrucciones con múltiples operandos, explotando en todo lo posible el paralelismo de datos soportado por los microprocesadores.

En este capítulo trataremos de dar una visión global de esta tecnología y de los nuevos elementos arquitectónicos que introduce en las arquitecturas PowerPC, así como las ventajas de utilizar dicha tecnología.

3.2.1 Introducción

La tecnología AltiVec amplia las posibilidades de los microprocesadores PowerPC hasta el límite, ejecutando procesos de propósito general mientras concurrentemente realiza una intensa computación algorítmica y soporta un amplio ancho de banda para el direccionamiento de procesos de datos en un único chip.

AltiVec es un conjunto de instrucciones SIMD en coma flotante y enteros diseñado y en propiedad de Apple Computer, IBM y Motorola, y puesto en ejecución en las versiones de PowerPC incluyendo el G4 de Motorola y los procesadores G5 de IBM. AltiVec es una marca registrada en propiedad de Motorola, así que el sistema también es nombrado como **Velocity Engine** por Apple y **VMX** por IBM.

AltiVec era el sistema SIMD más potente en una Unidad de Procesamiento Central de un ordenador de sobremesa cuando fue introducido al final de los años 90. Comparado con sus contemporáneos (MMX de sólo números enteros de Intel, SSE en coma flotante, y los diversos sistemas de otros fabricantes RISC), AltiVec ofrecía más registros que se podían utilizar de más formas y funcionar mediante un conjunto de instrucciones mucho más flexible. Sin embargo, el sistema SIMD de cuarta generación de Intel, SSE2, introducido con el Pentium IV, tiene muchas funciones similares a las de AltiVec.

Tanto AltiVec como los registros de vector de 128 bits de SSE2 pueden representar:

- Dieciséis caracteres de 8 bit con o sin signo
- Ocho enteros cortos con signo o sin signo de 16 bits
- Cuatro enteros o cuatro variables en coma flotante en formato de 32 bits

Estos últimos proporcionan instrucciones de control de la caché CPU previstas para reducir al mínimo la contaminación de la caché al trabajar con flujos de datos.

También muestran diferencias importantes. Al contrario que SSE2, AltiVec soporta un tipo de datos especial de "pixel" RGB, pero no opera con floats de doble precisión, y no hay manera de mover datos directamente entre los registros escalares y los de vectores.

En armonía con el modelo de "cargar/almacenar" del diseño RISC del PowerPC, los registros vectoriales, como los registros escalares, sólo se pueden cargar desde la memoria, y almacenar en ella. Sin embargo, AltiVec proporciona un sistema mucho más completo de operaciones "horizontales" que trabajan a través de todos los elementos de un vector; las combinaciones permitidas de los tipos de datos y de las operaciones con éstos son mucho más completas.

Se proporcionan 32 registros vectoriales de 128 bits, comparado con los 8 de SSE y SSE2, y la mayoría de las instrucciones de AltiVec toman tres operandos de registro, comparado con sólo dos operandos registro/registro o registro/memoria en un IA-32.

Las versiones recientes de GCC, Compilador Visual Age IBM y otros compiladores proporcionan intrínsecos para acceder a las instrucciones de AltiVec directamente desde programas en C y C++. La clase dedicada al almacenamiento "vector" aparece para permitir la declaración de los tipos nativos del vector, por ejemplo, *"vector unsigned char foo;"* declara una variable de 128 bits llamada *"foo"* que contiene dieciséis elementos sin signo de 8 bit.

Las funciones intrínsecas sobrecargadas tales como *"vec_add"* emiten el código de operación apropiado basado en el tipo de los elementos del vector, y se obliga a cumplir un fuerte tipado. En contraste, los tipos de datos definidos por Intel para los registros SIMD de IA-32 declaran solamente el tamaño del registro del vector (128 o 64 bits), y en el caso de un registro de 128 bits si contiene números enteros o valores en la coma flotante. El programador debe seleccionar el intrínseco apropiado para los tipos de datos empleados, por ejemplo *"_mm_add_epi16(x, y)"* para añadir dos vectores que contienen ocho números enteros de 16 bits.

Apple es el principal cliente de AltiVec, y lo usa para acelerar aplicaciones multimedia como QuickTime e ITunes y programas de procesamiento de imágenes tales como Adobe Photoshop. AltiVec también ha trabajado en partes clave del Mac OS X de Apple, incluido el compositor de gráficos Quartz. Motorola ha provisto las unidades de AltiVec en todos sus ordenadores de sobremesa desde el G4. AltiVec también se utiliza en algunos sistemas embebidos para proporcionar un proceso de la señal digital con un extremadamente alto rendimiento.

IBM ha dejado a VMX constantemente fuera de sus sistemas propietarios POWER, que están pensados para ser empleados como aplicaciones de servidor donde no es muy útil. Sin embargo, el ordenador de escritorio más reciente PowerPC 970 (apodado G5 por Apple), incluye una unidad de alto rendimiento de AltiVec. Incluye dos unidades funcionales para permitir efectos superescalares; un VMX completo en una unidad, y un multiplicador/sumador en la otra.

3.2.2 Aplicaciones

La tecnología AltiVec es una poderosa herramienta para los desarrolladores de software que quieren añadir eficacia y velocidad a sus aplicaciones.

La tecnología de AltiVec amplía la arquitectura del sistema de instrucciones (ISA) del PowerPC.

La idea principal es el tratamiento en paralelo de un conjunto de datos agrupados en un vector. El orden de este agrupamiento viene determinado por la subdivisión que se hace de dicho vector, siendo 2 (quadwords), 4 (words), 8 (half-words) y 16 (bytes) las posibles subdivisiones. Es decir, las operaciones de la tecnología de AltiVec pueden desarrollar múltiples conjuntos de datos en una sóla instrucción.

La tecnología AltiVec soporta las siguientes aplicaciones de audio y vídeo:

- Comunicaciones:
  - Módems multicanal
  - Cifrado de datos: RSA
  - Software para módem: V.34, 56K
- Realidad Virtual.
- Voz sobre tecnología IP (VoIP). VoIP transmite voz como paquetes comprimidos de datos digitales a través de Internet.
- Concentradores de Acceso/DSLAMS.
- Reconocedores de discurso. El procesamiento del discurso permite el reconocimiento de voz para el uso en aplicaciones como asistencia telefónica y marcado automático.
- Procesamiento de la voz/sonido: G.711, G.721, G.723, G, 729A, y AC-3. El procesamiento de la voz se usa para mejorar la calidad de sonido sobre las líneas.
- Gráficos 2D y 3D: QuickDraw, OpenGL, VRML, juegos,etc.
- Audio de alta fidelidad: audio 3D,AC-3,audio HI-FI usando las unidades en punto flotante de AltiVec.
- Procesamiento de imágenes y vídeo: JPEG, filtros.
- Cancelación del eco. Se utiliza para eliminar el eco en llamadas largas.
- Alto ancho de banda para la transmisión de datos.
- MPEG-1,MPEG-2,MPEG-4, H.234.
- Entrada-salida continua del discurso en tiempo real: HMM, aceleración del Viterbi, algoritmos neuronales.
- Vídeo conferencia: H.261, H.263.
- Máquinas inteligentes.

Usando el paralelismo de SIMD, el funcionamiento se puede acelerar en los procesadores de PowerPC a un nivel que pueda permitir el proceso concurrente en tiempo real de unas o más secuencias de datos.

La tecnología de AltiVec se puede utilizar como extensión de los varios microprocesadores del RISC; sin embargo, aquí lo discutimos dentro del contexto de la arquitectura de PowerPC descrita como sigue:

Modelo de programación

Tenemos dos características respecto al modelo de programación que detallamos a continuación:

- *Conjunto de instrucciones*. El repertorio de instrucciones AltiVec especifica las instrucciones que amplían el conjunto de instrucciones de PowerPC. Estas instrucciones están organizadas de una forma similar a las instrucciones de PowerPC (tales como load/store, instrucciones de vectores en punto flotante, instrucciones aritmético-lógicas, etc.).

- *Conjunto de registros*. El modelo de programación de AltiVec define los nuevos registros de AltiVec, los nuevos registros añadidos al conjunto de registros de PowerPC. Los registros existentes de PowerPC son afectados por la tecnología de AltiVec.

Modelo de memoria

La tecnología AltiVec especifica instrucciones adicionales para el manejo de la memoria caché. Es decir, un programa puede ejecutar las instrucciones de AltiVec que indican cuando una secuencia de unidades de memoria son alcanzables.

Modelo de las excepciones

Para asegurar la eficacia, la tecnología de AltiVec proporciona solamente una interrupción inasequible (VUI), una excepción de DSI, y la excepción de traza (si está implementada). No hay más excepciones que las excepciones DSI en las instrucciones load/store. Las instrucciones de AltiVec pueden causar las excepciones de PowerPC.

Modelo del manejo de la memoria

El modelo de memoria para la tecnología de AltiVec es igual que el implementado para la arquitectura de PowerPC. Los accesos de memoria de AltiVec se asumen siempre para ser alineados. Si un operando no está alineado, se utilizan las instrucciones adicionales de AltiVec para asegurarse de que está puesto correctamente en un registro del vector o en memoria.

3.2.3 Tecnología AltiVec

El empleo de la tecnología de AltiVec proporciona un acercamiento para acelerar el procesamiento de las secuencias de datos. Usando las instrucciones de AltiVec podemos alcanzar un aumento significativo de la velocidad en las comunicaciones, en programas multimedia y en todo el desarrollo de aplicaciones donde nos sea posible utilizar los diferentes niveles de paralelismo de datos. y reduciendo al mínimo el ancho de banda y los embotellamientos del acceso a memoria.

La tecnología de AltiVec amplía la arquitectura de PowerPC a través de la adición de una unidad de ejecución de vectores de 128 bits, que funciona simultáneamente con las unidades enteras y de punto flotante existentes. Esta nueva unidad de ejecución de vectores proporciona operaciones altamente paralelas, permitiendo la ejecución simultánea de operaciones múltiples en un solo ciclo de reloj.

Se puede pensar, la tecnología de AltiVec, como un sistema de registros y de unidades de ejecución que se pueden agregar a la arquitectura de PowerPC de una forma análoga a la adición de unidades de punto flotante. Estas últimas fueron agregadas para proporcionar una ayuda para los cálculos científicos de alta precisión, y ahora la tecnología de AltiVec se agrega a la arquitectura de PowerPC para acelerar los aspectos comentados anteriormente. A continuación se muestra como queda la arquitectura PowerPC con la tecnología AltiVec.

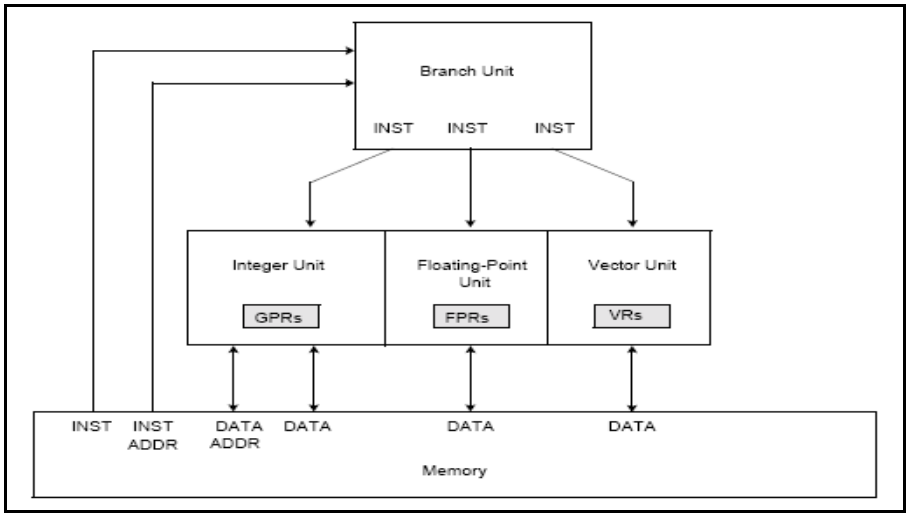

Figura 3.9 Estructura de la arquitectura PowerPC con AltiVec [9]

La tecnología AltiVec define lo siguiente:

- Longitud fija del vector a 128 bits que se puede subdividir en dieciséis bytes de 8 bits, ocho medias palabras (half words) de 16 bits, o cuatro palabras (words) de 32 bits cada una.
- Archivo de registros vectoriales (VRF) arquitectónicamente separado de los registros en punto flotante (FPRs) y de los registros de uso general (GPRs).
- Vector de enteros y aritmética en punto flotante.
- Cuatro operandos para la mayoría de las instrucciones (tres operandos de la fuente y un resultado).
- Selección de operaciones basadas en la utilización de algoritmos de procesamiento de señales digitales.
- Saturación. Es decir, los resultados sin signo se fijan a cero cuando hay underflow y al valor positivo máximo del número entero (2^n-1) cuando existe overflow. Para los resultados con signo, la saturación fija los resultados al número negativo representable más pequeño (- 2^{n-1}) cuando hay underflow, y al número positivo representable más grande (2^{n-1}-1) cuando hay overflow.
- Las instrucciones AltiVec proporcionan un vector de comparación y un mecanismo de selección para implementar ejecuciones condicionales como caminos preferidos para el control del flujo de datos en programas AltiVec.
- Interfaz realzado de la caché y la memoria.

3.2.4 Modelo arquitectónico AltiVec

A continuación definiremos de una manera general los siguientes aspectos relacionados con el modelo arquitectónico AltiVec:

- Registros y modelo de programación.
- Convenciones de los operandos.
- Formato de instrucciones y modos de direccionamiento.
- Modelo de la caché, excepciones y manejo de memoria.

3.2.4.1 Registros AltiVec y modelo de programación.

En la tecnología de AltiVec, la ALU está preparada para tener de uno a tres vectores fuente y producir un solo resultado/vector en cada instrucción.

La ALU es una unidad aritmética del estilo de SIMD que realiza la misma operación en todos los elementos de datos que abarquen cada vector. Este esquema permite programar código eficiente en un procesador altamente paralelo. Las instrucciones load/store son las únicas instrucciones que transfieren datos entre los registros y la memoria.

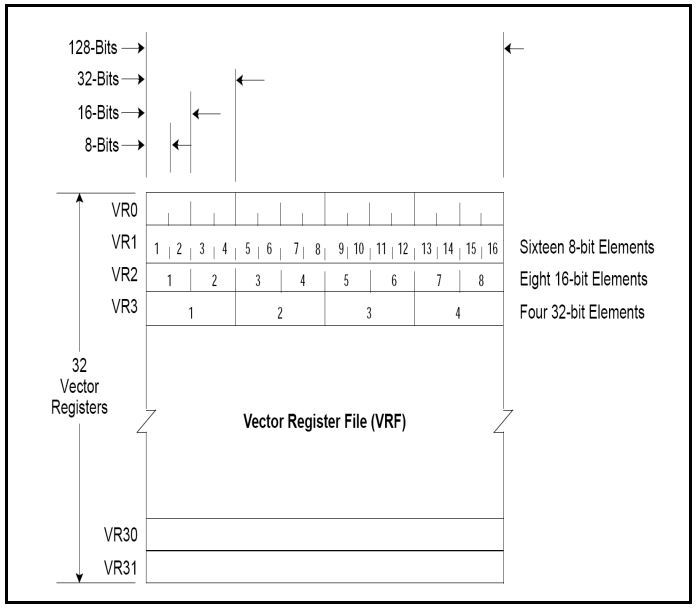

Figura 3.10 Unidad vectorial y archivo de registros vectoriales [2]

Arquitectónicamente, el archivo de registros vectoriales (VRF) está separado de los registros de propósito general y los registros de punto flotante. El modelo de programación de AltiVec incorpora los 32 registros del VRF, cada registro es de128 bits.

3.2.4.2 Convenciones de operandos

Las convenciones del operando definen cómo los datos del vector se almacenan en los registros y en la memoria.

3.2.4.3 Clasificación de bytes

La clasificación de los bytes por defecto para la arquitectura del repertorio de instrucciones de AltiVec es un PowerPC *big-endian,* pero AltiVec ofrece la posibilidad de operar en el modo *big-endian* o *little-endian*.

La arquitectura de PowerPC utiliza el registro de estado de la máquina (MSR) para especificar la clasificación de los bytes modo *little-endian* (LE). El MSR [LE] especifica el modo *endian* en el cual el procesador está funcionando actualmente.

Un valor 0 especifica modo *big-endian* y un valor de 1 especifica modo *little-endian*.

En el modo *big-endian* el soporte de la arquitectura de PowerPC no trata ningún elemento de datos más grande que una palabra doble, y la unidad básica de la memoria para los vectores es una quadword.

La clasificación *big-endian* para un registro vectorial se muestra a continuación. Los elementos de los registros vectoriales son numerados usando la clasificación *big-endian*. Por ejemplo, el byte más significativo está numerado con el 0, mientras que el menos significativo o el byte de la parte baja del vector es numerado con el 15.

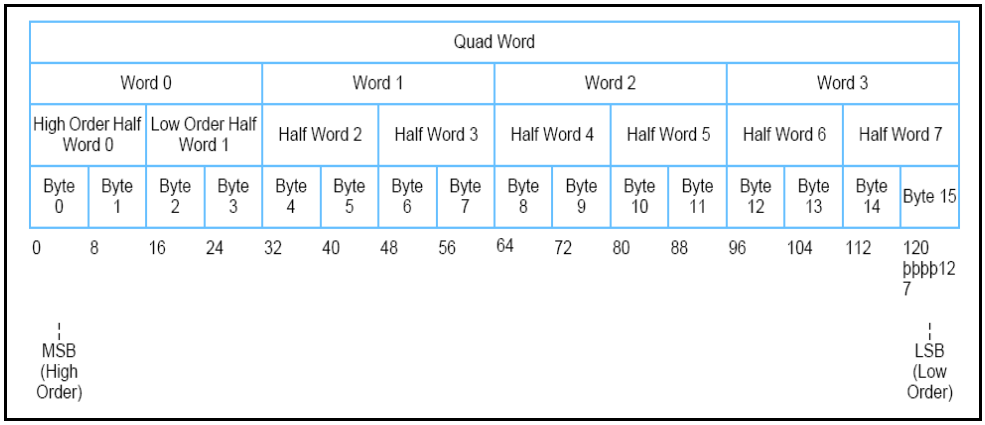

Figura 3.11 Clasificación *big-endian* para un registro vectorial [2]

En el modo *big-endian*, una instrucción de carga de quadword para la cual el direccionamiento efectivo (EA) está quadword alineado pone el byte tratado por el EA en el elemento 0 del byte del registro del vector destino. El byte tratado por EA+1 se pone en el elemento 1 del byte, y así sucesivamente. De manera análoga, una instrucción *store* de quadword para la cual el EA está quadword alineado pone el elemento 0 del byte del registro del vector de la fuente en el byte tratado por el EA. El elemento 1 del byte se pone en el byte tratado por EA+1, y así sucesivamente.

A continuación mostramos un ejemplo de como quedan en una quadword sus bytes en memoria.

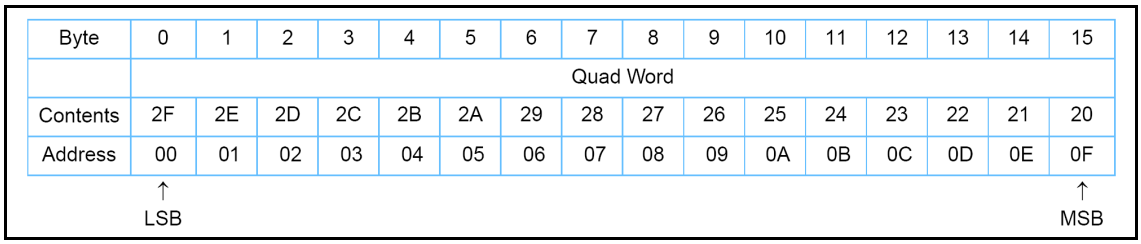

Figura 3.12 Clasificación *big-endian* para una quadword [2]

3.2.4.4 Accesos alineados y mal alineados

Los vectores son accedidos desde memoria con instrucciones tales como la carga del vector indexada (lvx) y la instrucción de almacenamiento indexada (stvx). El operando de un registro vectorial para una instrucción de acceso a memoria tiene un límite de alineación natural igual a la longitud del operando. Es decir, la dirección natural de un operando es un múltiplo integral de la longitud del operando. Un operando de la memoria se dice que está alineado si se alinea en su límite natural; si no se alinea mal. Las instrucciones de AltiVec son cuatro bytes largos y están palabra-alineados como las instrucciones de PowerPC.

Los operandos de los registros vectoriales en las instrucciones de acceso de memoria tienen las características mostradas en la tabla siguiente.

Operand	Length	32-bit Aligned Address (28-31)	64-bit Aligned Address (60-63)
Byte	8 bits (1 byte)	xxxx	xxxx
Half word	2 bytes	xxx0	xxx0
Word	4 bytes	xx00	xx00
Quad word	16 bytes	0000	0000
Note: An x in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.			

Figura 3.12 Alineación de los operandos [2]

El concepto de la alineación también se aplica más generalmente a los datos en memoria. Por ejemplo, un dato de 8 bytes se dice que es half-word alineado si su dirección es un múltiplo de dos; es decir, el direccionamiento efectivo (EA) señala al direccionamiento efectivo siguiente que es 2 bytes (media palabra) más allá del direccionamiento efectivo actual, que sería el EA+2 bytes, y entonces la siguiente sería EA+4 bytes, y el direccionamiento efectivo continuaría saltando cada 2 bytes (2 bytes = media palabra). Esto asegura que el direccionamiento efectivo es half-word alineado.

Es importante entender que los operandos de la memoria de AltiVec están entendidos para ser alineados, y los accesos de memoria de AltiVec están desarrollados como si el número apropiado de bits de peso más bajo del direccionamiento efectivo especificado fuera cero. Esta presunción es diferente del número entero de PowerPC y de las instrucciones de acceso en punto flotante de memoria donde la alineación no se asume siempre.

Para AltiVec ISA, el bit de peso más bajo del direccionamiento efectivo es ignorado para las instrucciones de acceso a memoria AltiVec del half-word, y los cuatro bits de orden inferior del direccionamiento efectivo son ignorados para las instrucciones de acceso a memoria AltiVec de quad-word. El efecto es cargar o almacenar el operando de la memoria de la longitud especificada que contiene el byte tratado por el direccionamiento efectivo.

Si se alinea mal un operando de la memoria, las instrucciones adicionales se deben utilizar para poner correctamente el operando en un registro vectorial o en memoria. La tecnología de AltiVec proporciona instrucciones para mover y combinar el contenido de dos registros vectoriales. Estas instrucciones facilitan el copiado de operandos mal alineados quad-word entre la memoria y los registros vectoriales.

3.2.4.5 Registro vectoriales y alineación del acceso de memoria

Cuando cargamos un byte alineado, half word, o una palabra de la memoria en un registro vectorial, el elemento que recibe los datos es el elemento que habría recibido los datos si tuviese la palabra alineada entera quad-word conteniendo el operando de la memoria apuntado por el direccionamiento efectivo cargado.

La posición del elemento en el registro vectorial destino o de la fuente depende del modo *endian*, como se describió anteriormente. (Los operandos de la memoria del byte se alinean siempre.)

Para el byte alineado, half-word, y las palabras de memoria, si se sabe el número correspondiente del elemento cuando se escribe el programa, las instrucciones apropiadas como *splat* y *permute* pueden ser utilizadas para copiar o replegar los datos contenidos en el operando de la memoria después de que cargue el operando en un registro vectorial. Las instrucciones de *splat* tomarán el contenido de un elemento en un registro y replicará dicho elemento en cada elemento del vector destino. La instrucción permute es la concatenación del contenido de dos vectores.

Otro ejemplo es replegar el elemento a través de un registro entero del vector antes de almacenarlo en un operando alineado arbitrario de la memoria de la misma longitud; la réplica se asegura que los datos correctos estén almacenados sin importar la compensación del operando de la memoria en su palabra alineada quad-word en memoria.

Los programadores deben tener mucho cuidado para la correcta alineación de los datos.

3.2.4.6 Alineamiento de datos quad-word

El AltiVec ISA no prevé las excepciones de la alineación para la carga y almacenamiento de los datos. Cuando ejecutamos un *load* o un *store*, el efecto es como si los cuatro bits de orden inferior de la dirección sean 0x0, sin importar el direccionamiento efectivo real generado. Puesto que los vectores pueden a menudo estar mal alineados debido a la naturaleza del algoritmo, AltiVec ISA proporciona la ayuda para el post-alineamiento de las cargas del quad-word y el pre-alineamiento para los almacenamientos del quad-word.

A continuación se observa que el efecto del intercambio descrito arriba se asume y que la disposición de los datos en el mapa lógico de memoria también falla.

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	Quad Word HI																Quad Word LO															
Contents				20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F													
Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
	MSB																														LSB	

Figura 3.13 Vector mal alineado en modo *big endian* [2]

3.2.4.7 Convenciones para punto flotante

El AltiVec ISA tiene básicamente dos modos para los punto flotante, uno es Java™/IEEE/C9X-modo obediente o un modo posiblemente más rápido no-Java/no-IEEE. Nos conformamos con la especificación del lenguaje Java 1, que es un subconjunto del entorno especificado por el estándar IEEE. Para los aspectos del comportamiento del punto flotante que no son definidos por Java sino que son definidos por el estándar de IEEE, AltiVec ISA se conforma con el estándar de IEEE. Para los aspectos del comportamiento en punto flotante que no son definidos ni por Java ni por el estándar de IEEE pero definidos por la propuesta punto flotante de C9X, WG14/N546 X3J11/96-010 AltiVec ISA se conforma con C9X cuando esta en modo Java-obediente.

3.2.4.8 Modo Java

El modo Java requiere solamente la conformidad con un subconjunto del estándar de Java/IEEE/C9X. Las ayudas del subconjunto de Java simplifican las implementaciones en punto flotante como sigue:

- Reduciendo el número de las operaciones que deben ser soportadas.
- Eliminando los indicadores de excepciones.
- Requiriendo solamente el modo de *"redondeo al más cercano"*, eliminando modos de redondeo dirigidos y redondeos asociados a señales de control.

La conformidad de Java requiere los siguientes aspectos del estándar IEEE:

- Soportar números denormalizados como entradas y resultados.
- Proveer de resultados NaN para las operaciones inválidas.
- Las comparaciones NaN con cualquier cosa dan siempre resultado falso.

3.2.4.9 Modo No Java

El intento de este modo es dar a los programadores una manera de asegurar el grado óptimo, respuesta dato-insensible, en tiempo real a través de la implementación. Otra manera para mejorar el tiempo de reacción sería implementar operaciones denormalizadas con la emulación del software.

Esto es arquitectónicamente posible,pero está bastante desaconsejado en implementaciones de modo No Java.

3.2.4.10 Excepciones punto flotante

Las excepciones punto flotante siguientes pueden ocurrir durante la ejecución de las instrucciones punto flotante de AltiVec:

División por cero	Operación inválida	Overflow
Logaritmo de cero	Operando NaN	Underflow

3.2.4.11 Modos de direccionamientos AltiVec

Como con las instrucciones de PowerPC, las instrucciones de AltiVec se codifican como instrucciones de palabras simples (32-bit). Los formatos de instrucción son constantes entre todos los tipos de instrucciones, permitiendo descifrar en paralelo los accesos a los operandos. Esta longitud de instrucción fija y formato constante simplifica la tubería de la instrucción. El load/store de AltiVec, y las instrucciones del prefetch utilizan códigos de operación secundarios y en el código de operación primario contienen 31 (0b011111). Todas las instrucciones de AltiVec del tipo ALU utilizan el código primario 4 (0b000100).

AltiVec ISA soporta operaciones de tipo *intraelementos* y de tipo *interelementos*. En una operación de tipo *intraelementos*, los elementos trabajan en paralelo sobre los elementos correspondientes de los múltiples registros de los operandos fuente y ponen los resultados en los campos correspondientes del registro del operando destino. Un ejemplo de una operación del *intraelementos* es el vector que suma palabras saturadas con signo (vaddsws) como muestra la siguiente figura.

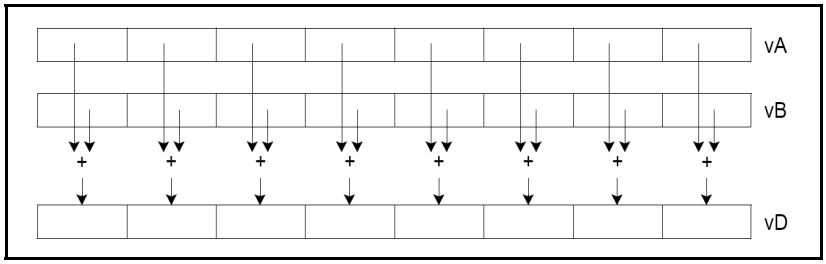

Figura 3.14 Ejemplo *intraelementos (vaddsws)* [2]

En este ejemplo, los cuatro elementos del entero con signo (32 bits) del registro vA se suman a los cuatro elementos correspondientes del entero con signo (32 bits) del registro vB y los cuatro resultados se ponen en los elementos correspondientes en el registro destino vD.

En operaciones *de tipo interelementos* los datos se mueven en cruz. Es decir, diversos elementos de cada operando fuente se utilizan para formar el operando de destino. Un ejemplo de una operación de tipo *interelementos* es el vector que realiza una permutación (vperm), que se muestra a continuación.

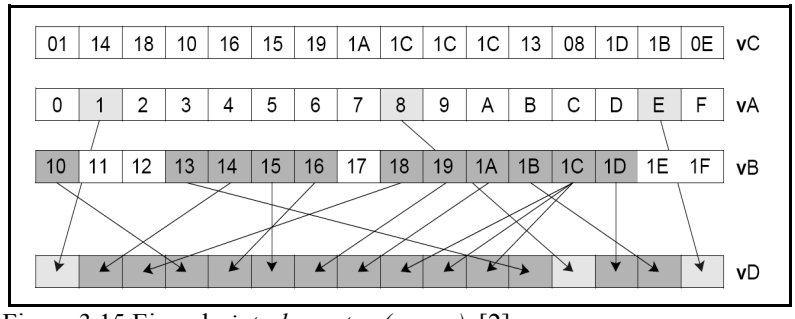

Figura 3.15 Ejemplo *interlementos (vperm)* [2]

En este ejemplo, *vperm* permite que cualquier byte dos registros del vector de la fuente (vA y vB) sea copiado a cualquier byte en el registro del vector destino, vD. En este caso los elementos de los registros del vector de la fuente no tienen elementos correspondientes que operen sobre el registro de destino.

La mayoría de las instrucciones aritmético-lógicas son operaciones de tipo *intraelementos.*

3.2.4.12 Modelo de la caché AltiVec

El AltiVec ISA define varias instrucciones para un mejor manejo de memoria. Estas instrucciones permiten que el software indique al hardware de la caché cómo debe hacer el *prefetch* y cómo se debe priorizar la escritura de datos (*writeback*). El AltiVec ISA no define aspectos del hardware de la implementación de la caché.

3.2.4.13 Modelo de excepciones AltiVec

La unidad del vector de AltiVec nunca genera una excepción. Las instrucciones nunca causarán una excepción por ellos mismos. Por lo tanto, en ningún momento causaría una excepción una instrucción load/store normal, tal como una violación de la protección o un fallo de página, la instrucción no toma una excepción de DSI; en su lugar, aborta y no se hace caso simplemente. La mayoría de las instrucciones de AltiVec no generan ninguna clase de excepción.

La unidad de AltiVec no divulga las excepciones de IEEE; no hay banderas del estado y la unidad no tiene ninguna trampa arquitectónica visible. Los resultados por defecto se producen para todas las condiciones de excepción según lo especificado primero por la especificación de Java. Si no existe ninguna por defecto, se utiliza estándar de IEEE (C9X).

3.2.4.14 Modelo para la gestión de la memoria

En el procesador de PowerPC las funciones primarias del MMU son que se ocupan de traducir direcciones (eficaces) lógicas a direcciones físicas para los accesos de memoria y los accesos de la entrada/salida (la mayoría de los accesos de la entrada/salida son asumidos por el mapa de memoria), y proporcionar la protección del acceso sobre un bloque o una página.
El AltiVec ISA no proporciona ninguna instrucción adicional al modelo de la gerencia de la memoria de PowerPC, pero las instrucciones de AltiVec tienen opciones para asegurarse de que un operando está puesto correctamente en un registro del vector o en memoria.

4 DESARROLLO DEL PROYECTO

El objetivo del proyecto es implementar un subconjunto de estas instrucciones y evaluar como influye dicha implementación en el rendimiento, utilizando para ello un conjunto de programa multimedia reales (MiBench, EEMBC, etc.).

El desarrollo del proyecto se llevará a cabo sobre un simulador de PowerPC (IBM/Freescale), al que se le irán añadiendo instrucciones del conjunto AltiVec.

4.1 Modificaciones del código

Para el desarrollo de nuestro proyecto se ha necesitado la construcción de nuevos tipos de tamaño 128 bits, así como la ampliación del campo de los registros necesarios para la implementación de las instrucciones AltiVec.

4.1.1 Tipos

Construimos una estructura nueva que nos permita manejar con facilidad los registros vectoriales de 128 bits o cualquier estructura de este tamaño.

4.1.1.1 Tipos genéricos

Primero vamos a mostrar los tipos definidos de antemano para los registros ya existentes.

Bits	Tipo	typedef
8	booleano	**int** bool_t
8	byte sin signo	**unsigned char** byte_t
8	byte con signo	**signed char** sbyte_t
16	half-word sin signo	**unsigned short** half_t
16	half-word con signo	**signed short** shalf_t
32	word sin signo	**unsigned int** word_t
32	word con signo	**signed int** sword_t
32	float de precisión simple	**float** sfloat_t
64	float de doble precisión	**double** dfloat_t
64	quad-word sin signo	**unsigned long long** qword_t
64	quad-word con signo	**signed long long** sqword_t

4.1.1.2 Tipos nuevos

Para poder añadir los registros vectoriales al código, primero hemos tenido que crear un tipo acorde a los distintos modos de acceso a dichos registros. La forma de acceder a estos registros nos encamina a crear un tipo definido *union* para poder acceder en cada instrucción a la parte del dato que necesitemos en ese momento.

```
typedef union {
        sqword_t lli[2];
        sword_t li[4];
        shalf_t si[8];
        sbyte_t bi[16];
        sfloat_t fi[4];
} vec_t;
```

Figura 4.1 Nuevo tipo de 128 bits

El nuevo tipo vendrá formado por varios arrays de diferentes tamaños, y dependiendo del tamaño del dato al que haga referencia, variará también el numero de elementos del array.

Asi, podremos trabajar con:

- Un registro formado por dos arrays de 64 bits con signo.
- Un registro formado por cuatro arrays de 32 bits con signo.
- Un registro formado por ocho arrays de 16 bits con signo.
- Un registro formado por dieciséis arrays de 8 bits con signo.
- Un registro formado por 4 arrays de 32 bits en punto flotante con signo.

4.1.2 Registros

Las características de la arquitectura AltiVec nos obliga a incluir 32 registros vectoriales de 128 bits que se usarán para la ejecución de las instrucciones, y 2 registros de control que se utilizarán para guardar y restaurar el estado.

4.1.2.1 Registros PowerPC

Los registros de la arquitectura PowerPC se pueden dividir entre los de acceso en modo supervisor y los de acceso en modo usuario. Serán estos últimos los que nosotros tratemos.

Primero vemos los registros de la arquitectura PowerPC en general y luego veremos los registros que hemos necesitado añadir para las operaciones AltiVec.

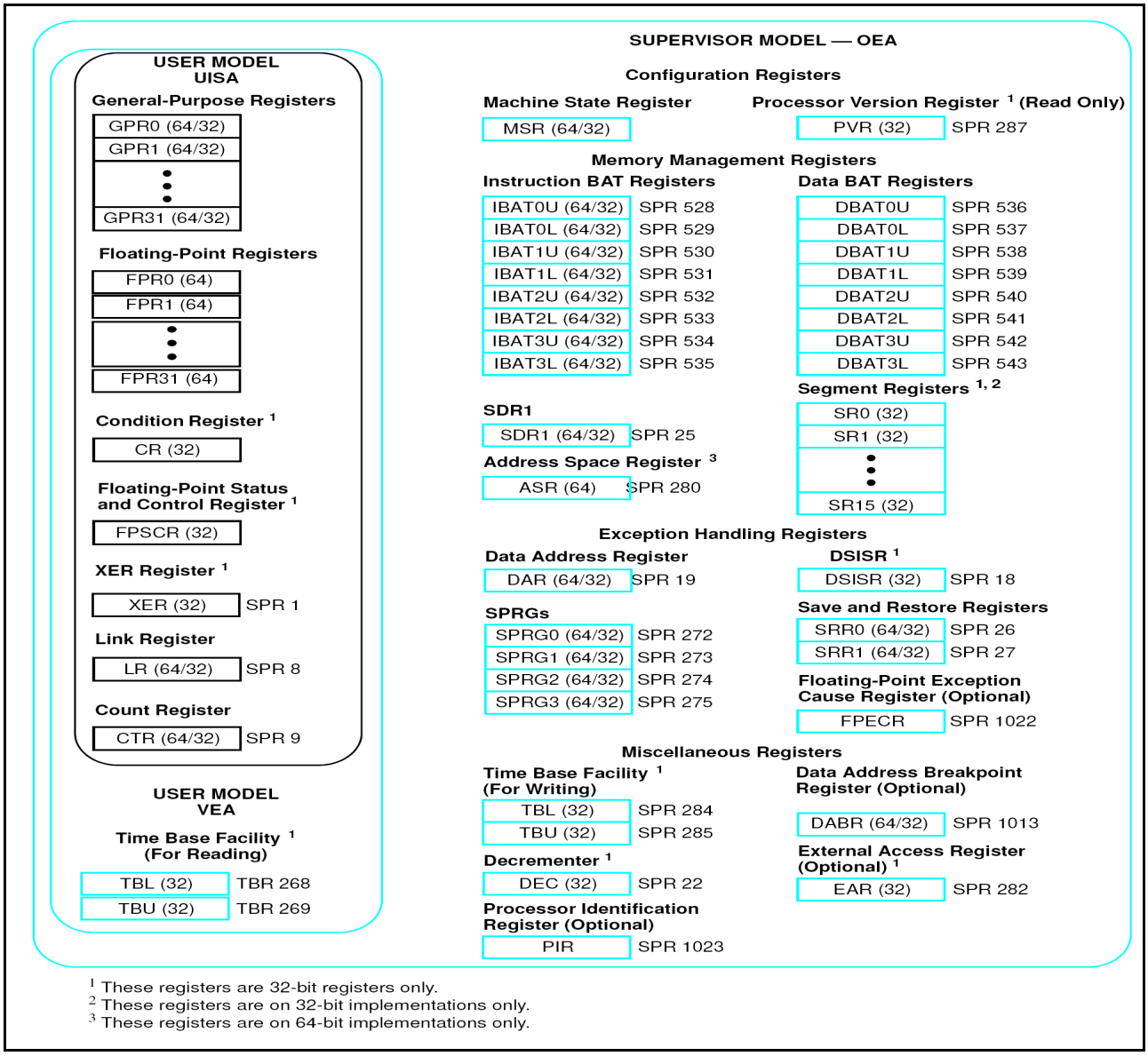

Figura 4.2 Vista general de la arquitectura PowerPC [1]

Estos registros son:

- Registros de Propósito General (GPR), que son 32 registros de 32 bit en las arquitecturas de 32 bits y de 64 en las de 64. Almacenan elementos de tipo entero. Estos registros se utilizan tanto de fuente como de destino en la sintaxis de la instrucción.

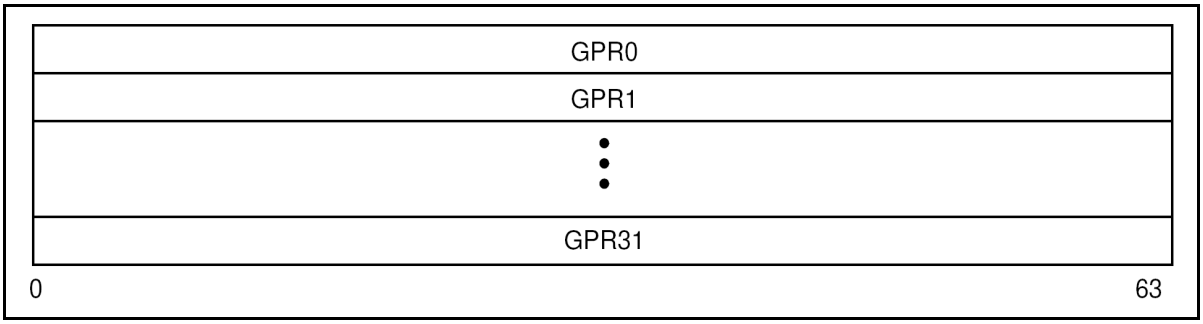

Figura 4.3 Registros de propósito general [1]

- Registros de Punto Flotante (FPR), los cuales son 32 registros de 64 bits que almacenan datos en punto flotante. Al igual que en los de propósito general a estos registros se accede como fuente y como destino, pero en instrucciones de punto flotante. Estos registros soportan el formato de doble precisión de las punto flotante. La información sobre el estado de las operaciones en punto flotante se almacena en el registro FPSCR (Floating-Point Status and Control Register), y en algunos casos en el registro CR (Condition Register) después de que se complete la instrucción.

  Las instrucciones de load y store de doble precisión en punto flotante transfieren datos de 64 bits entre la memoria y los registros sin realizar ninguna conversión.

  Las instrucciones de load con precisión simple en cambio tienen que convertir los datos a doble precisión para realizar la transferencia. En su caso, las instrucciones de store que leen doble precisión y lo guardan como precisión simple tienen que transformar el dato.

  Las instrucciones aritméticas en punto flotante de precisión simple o doble aceptan el formato de precisión doble de los registros FPR. Estas instrucciones generan un resultado intermedio, el cual deberá ser normalizado o no.

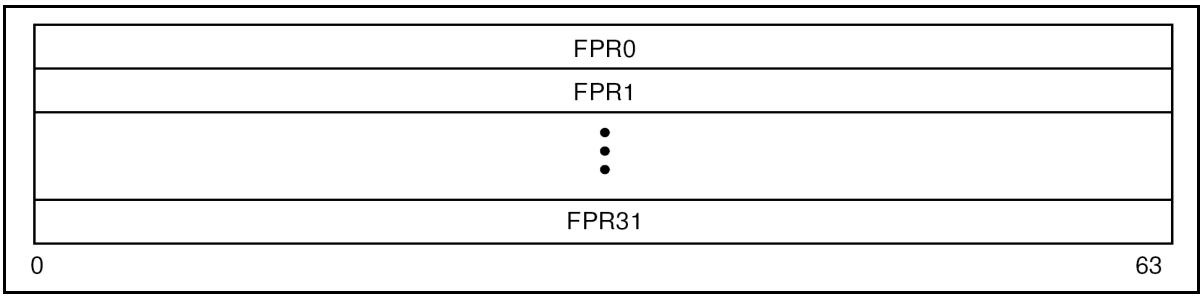

Figura 4.4 Registros punto flotante [1]

- Registro Condición (CR), es un registro de 32 bits que refleja el resultado de algunas operaciones, y provee mecanismos de control. Los bits de este registro se agrupan en 8 campos de cuatro bits para facilitar su uso.

  CR0 : Este campo nos dice el signo de la operación.
  CR1 : Este campo guarda las excepciones de las operaciones en punto flotante.
  CRn : El resto de campos guardan información de comparación.

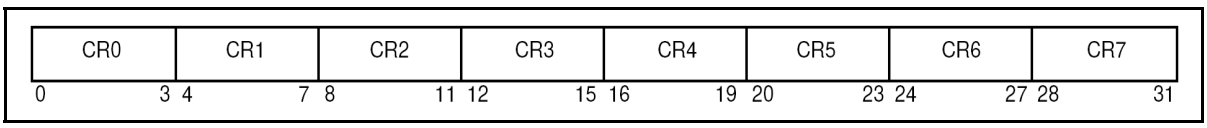

Figura 4.5 Registro condición (CR) [1]

- Registro de control y estado de punto flotante (FPSCR), es un registro de 32 bits que se encarga de guardar las excepciones generadas por las operaciones en punto flotante, guardar el tipo del resultado producido en las operaciones de punto flotante, controlar el modo de redondeo utilizado en dichas operaciones, así como habilitar o deshabilitar la información de excepciones.

Los primeros 24 bits son bits de estado, mientras que el resto son bits de control. Los bits de estado se actualizan al completarse la ejecución de la instrucción.

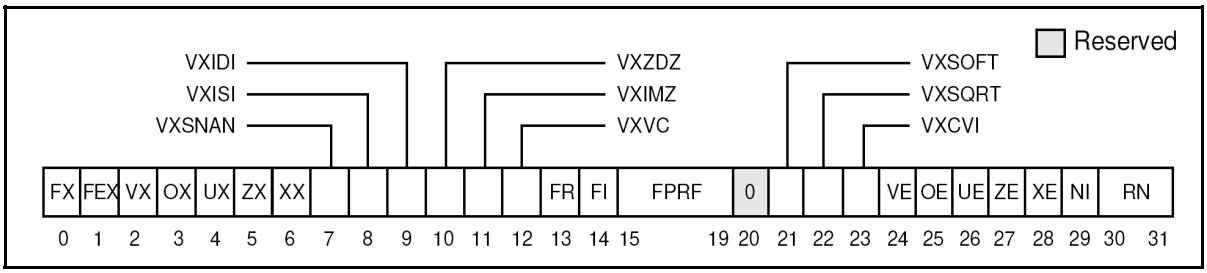

Figura 4.6 Registro de control [1]

- Registro XER, es un registro de 32 bits que guarda el resultado de operaciones con acarreo para saber si a habido overflow o acarreo.

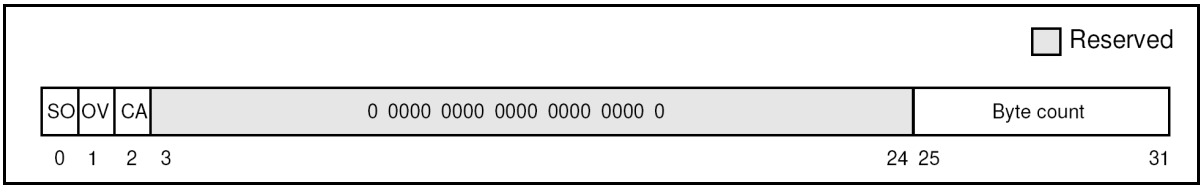

Figura 4.7 Registro XER [1]

- Registro LINK, este registro se utiliza para las instrucciones de salto.

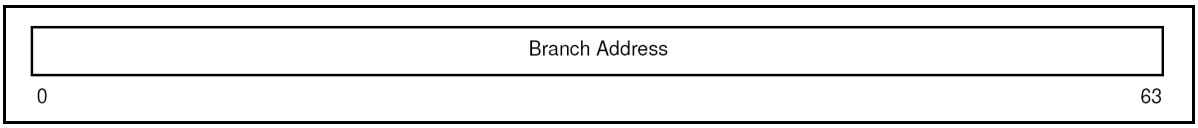

Figura 4.8 Registro LINK [1]

- Registro Contador, nos ofrece un bucle que se decrementa con la ejecución de instrucciones de salto apropiadas.

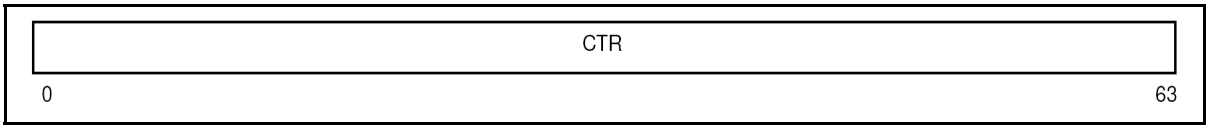

Figura 4.9 Registro contador [1]

4.1.2.2 Registros AltiVec

Al igual que con los registros generales de PowerPC, el conjunto de registros AltiVec también se puede acceder en modo usuario y en modo supervisor.

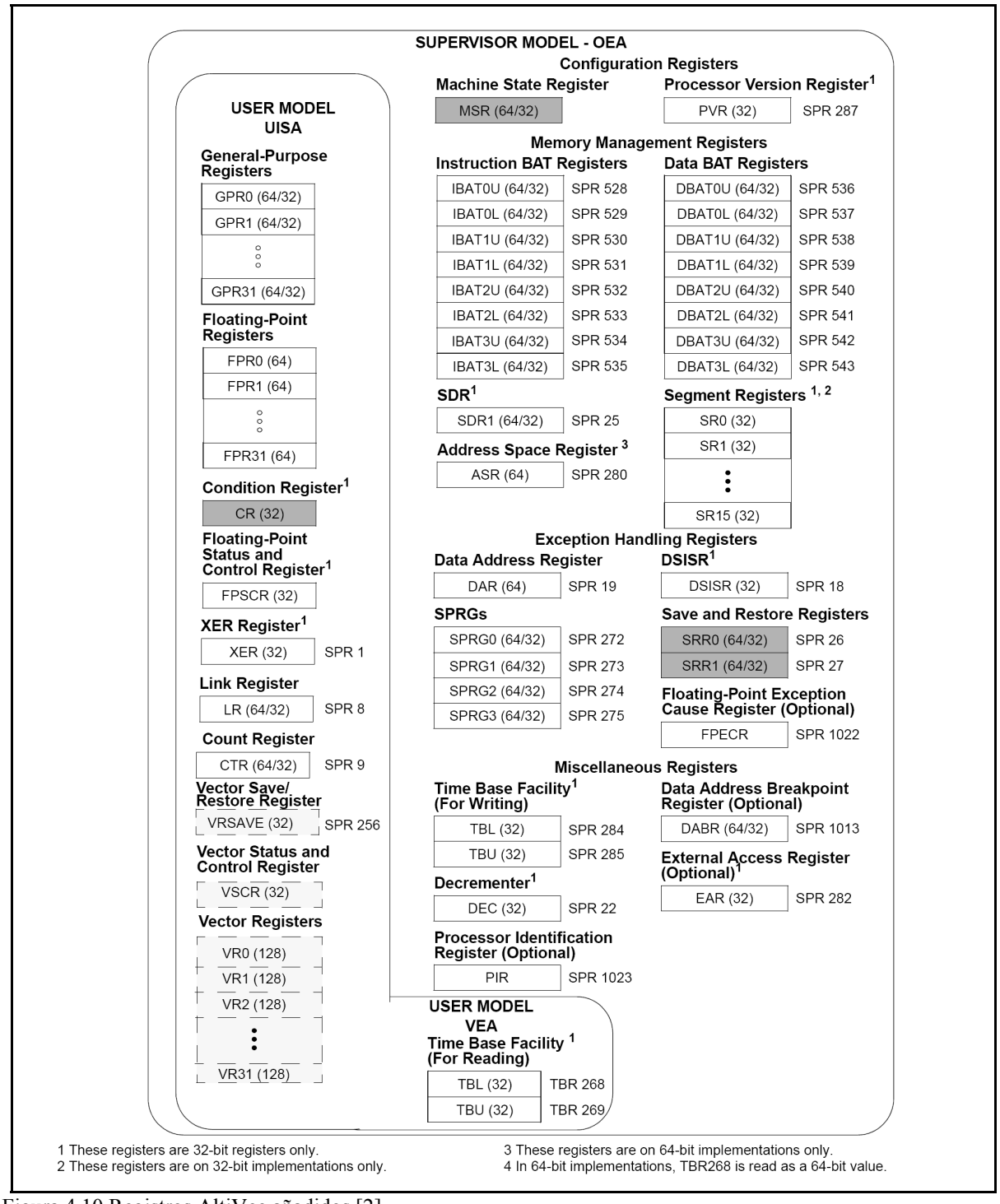

Figura 4.10 Registros AltiVec añadidos [2]

El conjunto de registros AltiVec está formado por los siguientes registros:

- Registros Vectoriales (VPR), son 32 registros de 128 bits cada uno. Cada registro vectorial se puede tomar por 16 elementos de 8 bits, 8 elementos de 16 bits ó 4 elementos de 32 bits. Los registros vectoriales son accedidos como operandos de una instrucción vectorial. Los accesos a los registros son explícitos como parte de la ejecución de una instrucción.

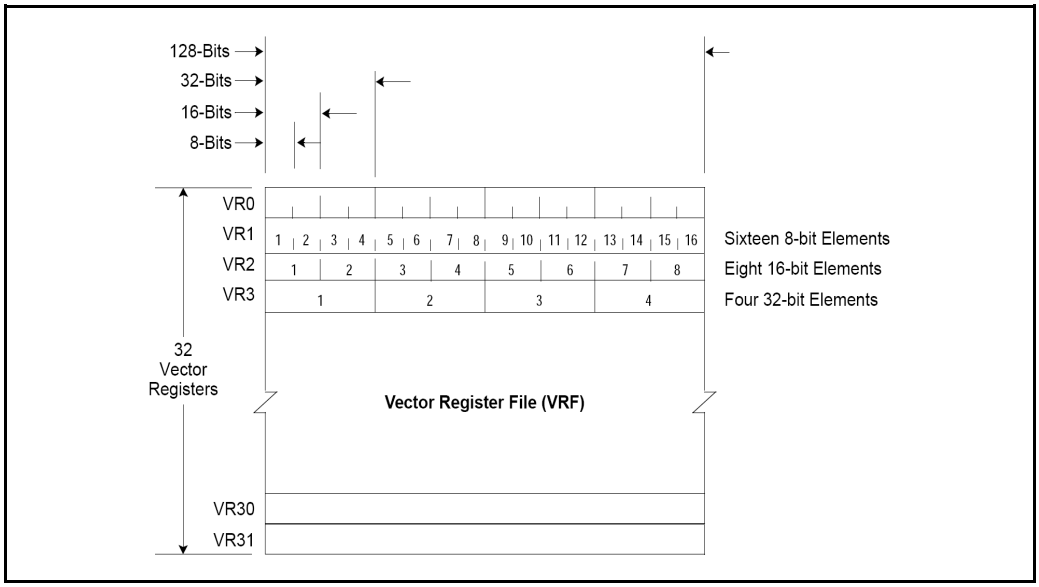

Figura 4.11 Registros vectoriales (VPR) [2]

- Registro de control y estado vectorial (VSCR), es un registro especial de 32 bits que funciona de forma similar al registro FPSCR de la arquitectura PowerPC.

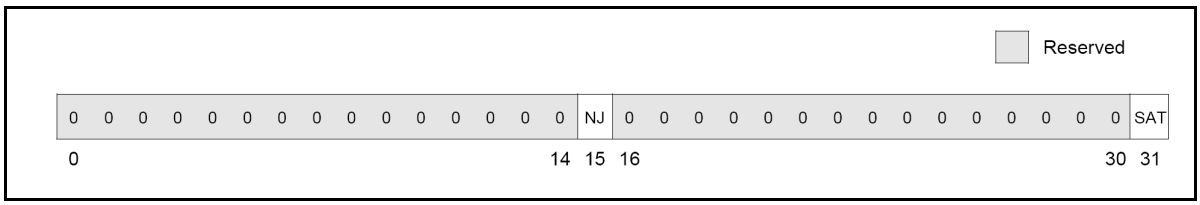

Figura 4.12 Registro de control VSCR [2]

El registro VSCR tiene dos bits definidos, el modo No-Java (NJ) y el de saturación. El resto de bits están reservados.

Hay dos instrucciones especiales que mueven el contenido del VSCR a un registro vectorial, o el contenido de un registro vectorial al registro VSCR. Cuando movemos a un registro vectorial limpiamos el contenido de los 96 bits que no usamos.

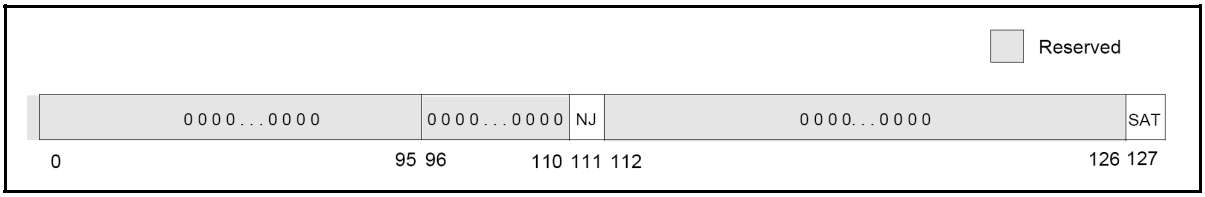

Figura 4.13 Bits modificados del registro VSCR [2]

- Registro VRSAVE, este registro de 32 bits, se utiliza para guardar y restaurar el estado. Cada bit hace referencia a un registro vectorial e indica si un registro concreto está siendo usado en la ejecución del proceso.

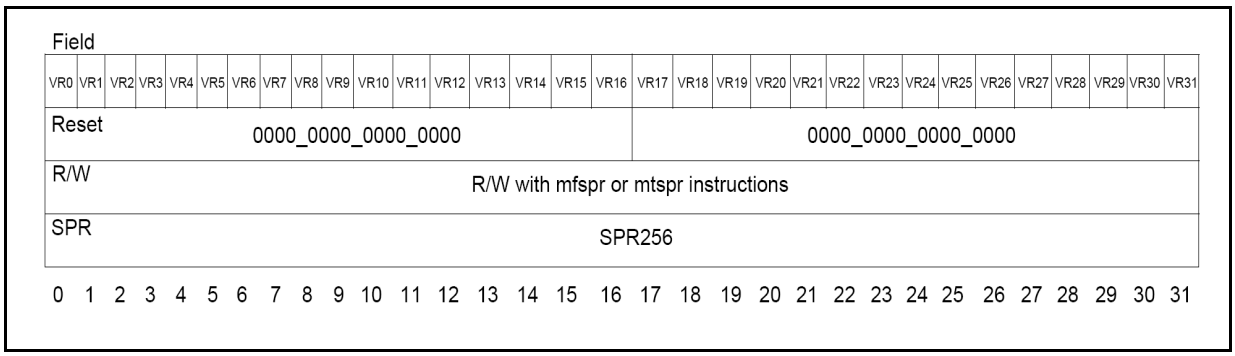

Figura 4.14 Registro de control VRSAVE [2]

- Registro Condición, para las instrucciones AltiVec se utiliza el campo CR6 del registro condición. Si en alguna instrucción AltiVec el bit Rc está marcado en instrucciones de comparación, el campo CR6 se actualiza.

Bit CR	Bit CR6	Comparación vectorial
24	0	1: La relación es cierta para todos los elementos (vD a 1)
25	1	0
26	2	1: La relación es falsa para todos los elementos (vD a 0)
27	3	0

4.1.3 Conjunto de instrucciones AltiVec

Las instrucciones AltiVec son de tamaño 32 bits. El conjunto de instrucciones de esta arquitectura presenta 4 operandos, siendo tres de ellos vectores fuente y uno el vector resultado. Los bits del 0-5 especifican el código de operación primario de las instrucciones AltiVec.

4.1.3.1 Formato de las instrucciones

Las instrucciones AltiVec presentan distintos formatos. Esto es debido a que algunas presentan códigos de operación primario y secundario, distinto número de operandos entre unas instrucciones y otras,etc.

Los diferentes formatos son:

- Instrucciones VA:

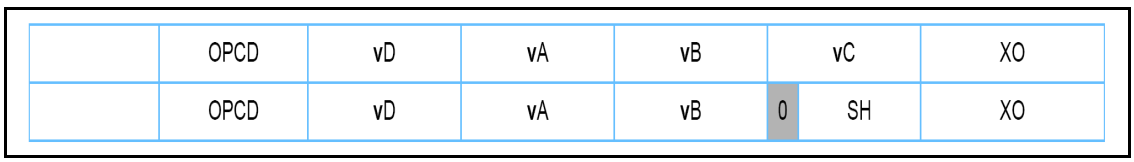

Figura 4.15 Formato de las VA [2]

- Instrucciones VXR:

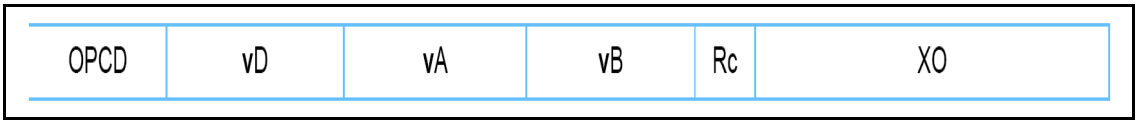

Figura 4.16 Formato de las VXR [2]

- Instrucciones VX:

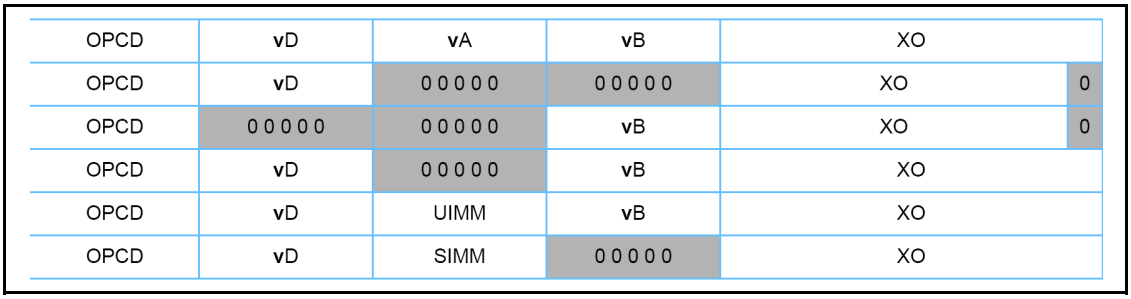

Figura 4.17 Formato de las VX [2]

- Instrucciones X

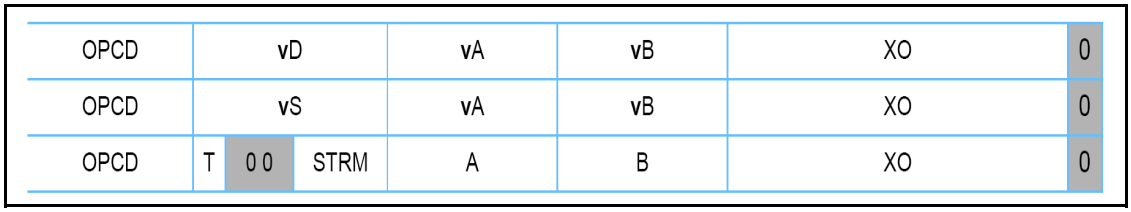

Figura 4.18 Formato de las X [2]

A continuación describimos el significado de los campos usados en los distintos formatos de instrucciones:

Campo	**Bits**	**Descripción**
OPCD	0-5	Código de operación primario
rA	11-15	Registro de propósito general usado como fuente o destino
rB	16-20	Registro de propósito general usado como fuente
Rc	31	0: No actualiza el registro de condición
		1: Actualiza el registro de condición para controlar el flujo del programa
vA	11-15	Registro vectorial usado como fuente
vB	16-20	Registro vectorial usado como fuente
vC	21-25	Registro vectorial usado como fuente
vD	6-10	Registro vectorial usado como destino
vS	6-10	Registro vectorial usado como fuente
SHB	22-25	Cantidad a desplazar en bytes
SIMM	11-15	Inmediato, usado para especificar un entero con signo
UIMM	11-15	Inmediato, usado para especificar un entero sin signo
XO	26-31	Código de operación extendido

Dentro del registro vectorial, un byte, media palabra o palabra son referenciados de la siguiente manera:

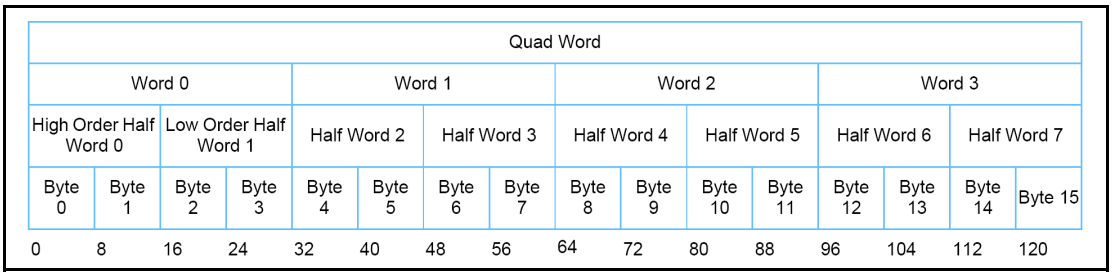

Figura 4.19 Posibles particiones de un registro vectorial [2]

4.1.3.2 Tipos de instrucciones

Las instrucciones AltiVec pueden ser agrupadas de la siguiente forma:

a) *Instrucciones vectoriales aritméticas con enteros.* Estas instrucciones están definidas por la UISA. Usan el contenido de los registros vectoriales como operandos fuente, y almacenan los resultados en los propios registros vectoriales.

   Las instrucciones vectoriales con enteros tratan a los operandos fuente como enteros con signo, a menos que la instrucción sea identificada explícitamente como el desarrollo de una operación sin signo. Incluyen las siguientes categorías de instrucciones:

   - Aritméticas
   - De comparación
   - Lógicas
   - De desplazamiento y rotación

b) *Instrucciones vectoriales con punto flotante*. Las instrucciones de punto flotante operan sobre operandos de precisión simple. El formato de los números en punto flotante se conforma con el modelo estándar ANSI/IEEE-754. Incluyen las siguientes categorías de instrucciones:

   - Aritméticas
   - De redondeo y conversión
   - De comparación
   - De estimación en punto flotante

c) *Instrucciones vectoriales de carga:* Para estas instrucciones, el byte, la media palabra, o la palabra direccionada por la dirección efectiva es almacenada en el registro destino. El modelo de ordenamiento es *big-endian*, como en la arquitectura PowerPC.

d) *Instrucciones vectoriales de almacenamiento:* Para estas instrucciones, el contenido del registro vectorial usado como operando fuente es almacenado en el byte, la media palabra, palabra, o doble palabra de la dirección de memoria indicada por la dirección efectiva.

A continuación vamos a detallar las instrucciones separándolas en diferentes grupos según la operación que realicen.

El formato que vamos a seguir va a ser el siguiente:

- Titulo de las instrucciones vectoriales.
- Tabla que contiene las instrucciones con un comportamiento similar.
- Descripción de su funcionamiento.
- Dibujo explicativo (opcional)

a) Instrucciones vectoriales aritméticas con enteros

a.1) Aritméticas.

Las instrucciones que abarcamos en este apartado son:

1. Suma y resta modular
2. Suma y resta saturada con y sin signo
3. Suma y resta de enteros sin signo y con acarreo
4. Multiplicación modular de enteros con y sin signo
5. Multiplicación alta y suma saturada con signo
6. Multiplicación baja y suma saturada sin signo
7. Multiplicación y suma modular (saturada) de enteros con y sin signo
8. Suma saturada a través de una palabra con signo
9. Suma saturada parcial a través de una palabra con y sin signo
10. Promedio de enteros con (sin) signo

1. Suma y resta modular

SUMA MODULAR	RESTA MODULAR
vaddubm	*vsububm*
vadduhm	*vsubuhm*
vadduwm	*vsubuwm*

- Cada elemento entero de vA es sumado o restado, según el caso, modularmente a su correspondiente elemento entero de vB.
- El entero resultante es colocado en el correspondiente elemento de vD.
- Esta instrucción puede ser usada tanto para números con signo como sin signo.

2. Suma y resta saturada con y sin signo

SUMA SATURADA Con signo	RESTA SATURADA Con signo	SUMA SATURADA Sin signo	RESTA SATURADA Sin signo
vaddsbs	vsubsbs	vaddubs	vsububs
vaddshs	vsubshs	vadduhs	vsubuhs
vaddsws	vsubsws	vadduws	vsubuws

Para las operaciones sin signo de suma y resta saturadas:

- Cada número entero sin signo de vA es sumado (restado) a su correspondiente número entero sin signo de vB.
- Si la suma (resta) resultante es mayor que 2^n-1, entonces el resultado se satura a 2^n-1, y activamos el bit de saturación.
- Como resultado obtendremos los números enteros sin signo parciales, que serán colocados en el correspondiente elemento de vD.

Para las operaciones con signo de suma y resta saturadas:

- Cada número entero con signo de vA es sumado (restado) a su correspondiente número entero con signo de vB.
- Si la suma (resta) resultante es mayor que 2^n-1, entonces el resultado se satura a 2^n-1, y si la suma (resta) resultante es menor que -2^{n-1}, entonces el resultado se satura a -2^{n-1}. Además, en caso de darse alguno de los dos casos anteriores, activamos el bit de saturación.
- Como resultado obtendremos los números enteros con signo parciales, que serán colocados en el correspondiente elemento de vD.

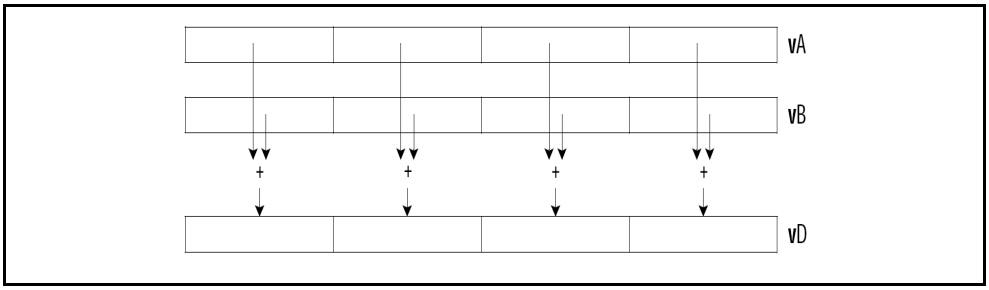

Figura 4.20 Instrucción VADDSWS [2]

3. Suma y Resta de enteros sin signo y con acarreo

SUMA SIN SIGNO	RESTA SIN SIGNO
vaddcuw	vsubcuw

- Cada número entero sin signo de vA es sumado (restado) a su correspondiente número entero sin signo de vB.
- Los acarreos de salida resultantes son correspondientemente colocados en vD

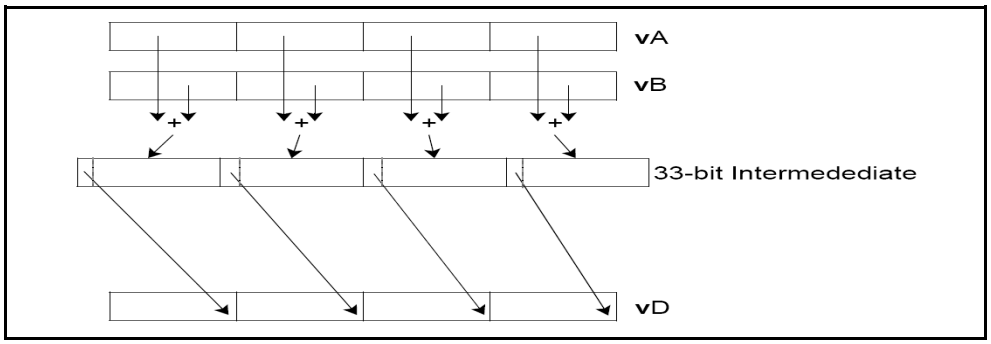

Figura 4.21 Instrucción VADDCUW [2]

4. Multiplicación modular de enteros con y sin signo

MULTIPLICACIÓN MODULAR DE ENTEROS SIN SIGNO	*MULTIPLICACIÓN MODULAR DE ENTEROS CON SIGNO*
vmuloub	vmulosb
vmulouh	vmulosh
vmuleub	vmulesb
vmuleuh	vmulesh

Para las operaciones de multiplicación modular sin signo:

- Cada número entero sin signo en posición impar (par) de vA es multiplicado modularmente a su correspondiente número entero sin signo en posición impar (par) de vB.
- El resultado, de tamaño doble, es almacenado en su correspondiente posición en vD.

Para las operaciones de multiplicación modular con signo:

- Cada número entero con signo en posición impar (par) de vA es multiplicado modularmente a su correspondiente número entero con signo en posición impar (par) de vB.
- El resultado, de tamaño doble, es almacenado en su correspondiente posición en vD.

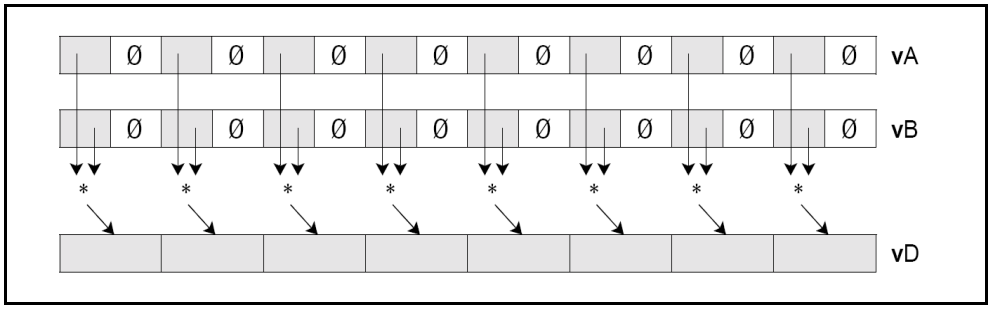

Figura 4.22 Instrucción VMULESB [2]

5. Multiplicación alta y suma saturada con signo

MULTIPLICACIÓN ALTA Y SUMA SATURADA CON SIGNO
vmhaddshs
vmhraddshs

- Cada número entero con signo de vA es multiplicado a su correspondiente número entero con signo de vB, produciendo como resultado intermedio un entero con signo de tamaño doble.
- La parte alta de este resultado intermedio (redondeado) se suma al correspondiente entero con signo (extendido) de vC.
- Si la suma resultante es mayor que 2^n-1, entonces el resultado se satura a 2^n-1, y si la suma es menor que -2^{n-1}, entonces el resultado se satura a -2^{n-1}. Además, en caso de darse alguno de los dos casos anteriores, activamos el bit de saturación.
- El resultado es almacenado en su correspondiente posición en vD.

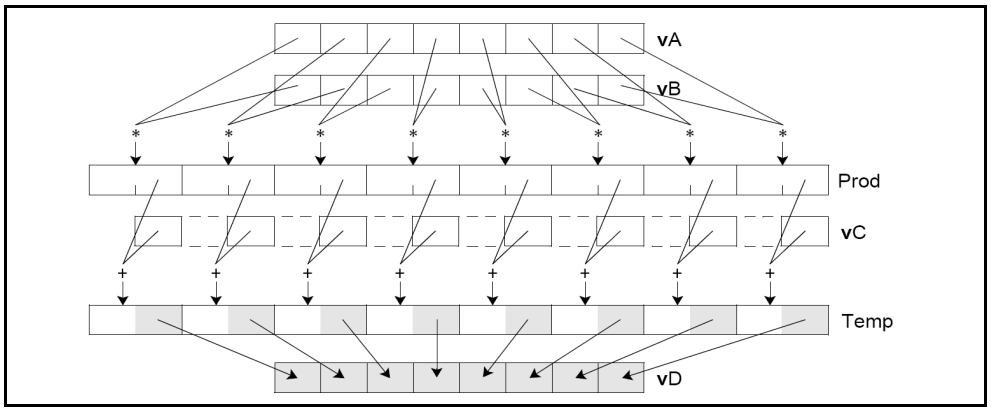

Figura 4.23 Instrucción VMHADDSHS [2]

6. Multiplicación baja y suma modular sin signo

MULTIPLICACIÓN BAJA Y SUMA MODULAR SIN SIGNO
vmladduhm

- Cada número entero de vA es multiplicado a su correspondiente número entero de vB, produciendo como resultado intermedio un entero de tamaño doble.
- La parte baja de este resultado intermedio se suma modularmente al correspondiente entero de vC.
- El resultado es almacenado en su correspondiente posición en vD.
- Observamos que esta instrucción puede ser usada tanto para números con signo como sin signo.

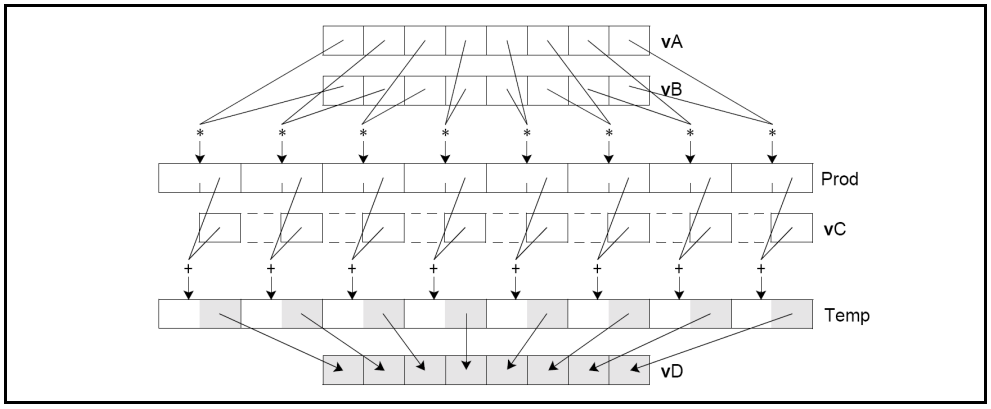

Figura 4.24 Instrucción VMLADDUHM [2]

7. Multiplicación y suma modular (saturada) de enteros sin signo

MULTIPLICACIÓN Y SUMA MODULAR (SATURADA) DE ENTEROS SIN SIGNO	MULTIPLICACIÓN Y SUMA MODULAR (SATURADA) DE ENTEROS CON SIGNO
vmsumubm	vmsumshm
vmsumuhm	vmsumshs
vmsumuhs	

Para las operaciones de multiplicación y suma modular saturada sin signo:

- Cada número entero sin signo de vA es multiplicado a su correspondiente número entero sin signo de vB, produciendo como resultado intermedio un entero de tamaño doble.
- A continuación realizamos la suma modular (saturada) de las n correspondientes sumas parciales y el correspondiente entero sin signo de vC.
- El resultado es un entero sin signo que almacenamos en su correspondiente posición en vD.

Para las operaciones de multiplicación y suma modular saturada con signo:

- Cada número entero con signo de vA es multiplicado a su correspondiente número entero con signo de vB, produciendo como resultado intermedio un entero con signo de tamaño doble.
- A continuación realizamos la suma modular (saturada) de las n correspondientes sumas parciales y el correspondiente entero con signo de vC.
- El resultado es un entero con signo que almacenamos en su correspondiente posición en vD.

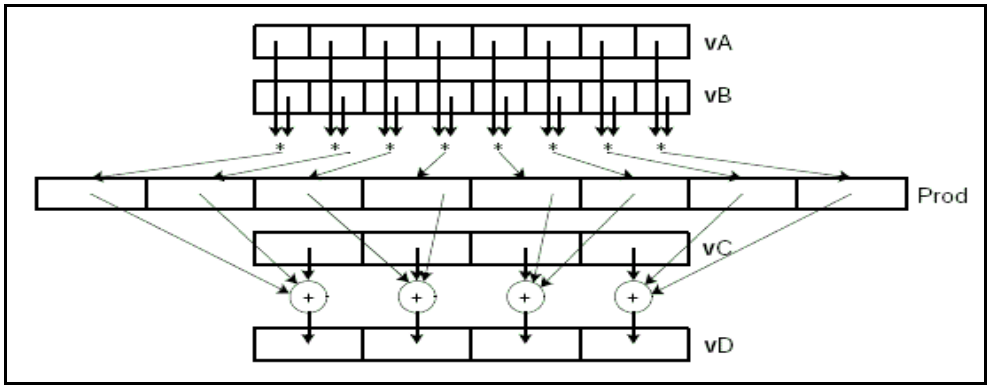

Figura 4.25 Instrucción VMSUMUHM [2]

La instrucción VMSUMMBM es un caso particular. Su comportamiento lo detallamos a continuación:

- Cada número entero con signo de vA es multiplicado a su correspondiente número entero sin signo de vB, produciendo como resultado intermedio un entero de tamaño doble.
- A continuación realizamos la suma modular de las n correspondientes sumas parciales y el correspondiente entero con signo de vC.
- El resultado es un entero con signo que almacenamos en su correspondiente posición en vD.

8. Suma saturada a través de una palabra con signo

SUMA SATURADA A TRAVÉS DE UNA PALABRA CON SIGNO
vsumsws

- Realiza la suma saturada de todos los elementos de vA y la última palabra de vB.
- Si se produce saturación, activamos el bit SAT.
- Limpiamos el contenido de vD, y almacenamos el resultado en la posición correspondiente.

9. Suma saturada parcial a través de una palabra con (sin) signo

SUMA SATURADA PARCIAL A TRAVÉS DE UNA PALABRA CON Y SIN SIGNO
vsum2sws
vsum4sbs
vsum4shs
vsum4ubs

- Realiza la suma saturada de la mitad (la cuarta parte) de los elementos de vA con su correspondiente palabra de vB.
- Si se produce saturación, activamos el bit SAT.
- El resultado lo almacenamos en la posición correspondiente en vD.

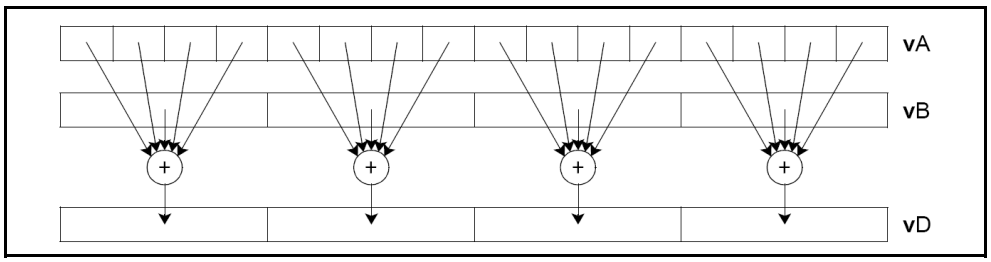

Figura 4.26 Instrucción VSUM4UBS [2]

10. Promedio de enteros con (sin) signo

PROMEDIO DE ENTEROS SIN SIGNO	PROMEDIO DE ENTEROS CON SIGNO
vavgub	vavgsb
vavguh	vavgsh
vavguw	vavgsw

- Cada elemento entero con (sin) signo de vA es sumado a su correspondiente elemento entero con (sin) signo de vB.
- A la suma parcial resultante le sumamos 1.
- El resultado lo almacenamos en la posición correspondiente en vD.

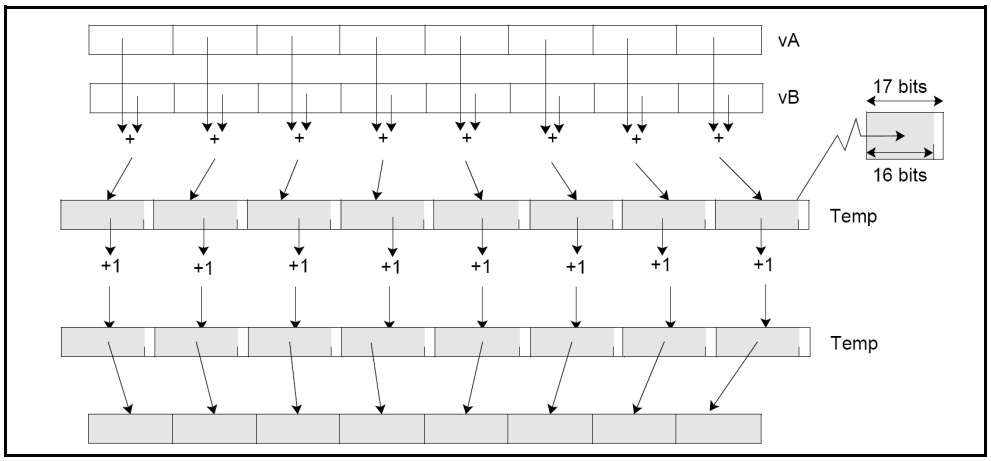

Figura 4.27 Instrucción VAVGUH [2]

a.2) De comparación.

Las instrucciones que abarcamos en este apartado son:

1. Máximo de enteros con y sin signo
2. Mínimo de enteros con y sin signo
3. *Mayor que e igual que* sin signo
4. *Mayor que e igual que* con signo

1. Máximo de enteros con y sin signo

MÁXIMO DE ENTEROS SIN SIGNO	MÁXIMO DE ENTEROS CON SIGNO
vmaxub	vmaxsb
vmaxuh	vmaxsh
vmaxuw	vmaxsw

- Cada elemento entero con (sin) signo de vA es comparado con su correspondiente elemento entero con (sin) signo de vB.
- El valor del máximo lo almacenamos en la posición correspondiente en vD.

2. Mínimo de enteros con y sin signo

MÁXIMO DE ENTEROS SIN SIGNO	MÁXIMO DE ENTEROS CON SIGNO
vminub	vminsb
vminuh	vminsh
vminuw	vminsw

- Cada elemento entero con (sin) signo de vA es comparado con su correspondiente elemento entero con (sin) signo de vB.
- El valor del mínimo lo almacenamos en la posición correspondiente en vD.

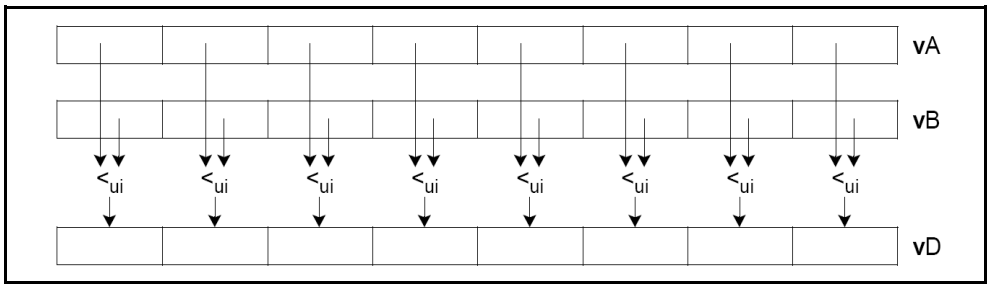

Figura 4.28 Instrucción VMINUH [2]

3. Mayor que e igual que sin signo

MAYOR QUE	IGUAL QUE
vcmpgtub*x*	vcmpequb*x*
vcmpgtuh*x*	vcmpequh*x*
vcmpgtuw*x*	vcmpequw*x*

- Cada elemento entero sin signo de vA es comparado con su correspondiente elemento entero sin signo de vB.
- El resultado de la comparación lo almacenamos en la posición correspondiente en vD.
- En caso de ser ciertas o falsas todos los pares de comparaciones, actualizamos el campo 6 del registro CR

4. M*ayor que e igual que con signo*

MAYOR QUE	IGUAL QUE
vcmpgtsbx	vcmpeqsbx
vcmpgtshx	vcmpeqshx
vcmpgtswx	vcmpeqswx

- Cada elemento entero con signo de vA es comparado con su correspondiente elemento entero con signo de vB.
- El resultado de la comparación lo almacenamos en la posición correspondiente en vD.
- En caso de ser ciertas o falsas todos los pares de comparaciones, actualizamos el campo 6 del registro CR.

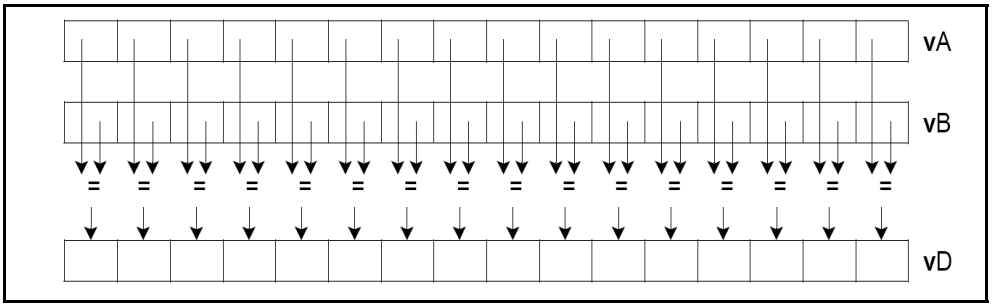

Figura 4.29 Instrucción VCMPEQUB [2]

a.3) Lógicas.

OPERACIONES LÓGICAS
vand
vor
vxor
vandc
vnor

- Realiza la operación lógica con cada elemento de vA y su correspondiente elemento de vB.
- El resultado lo almacenamos en la posición correspondiente en vD.

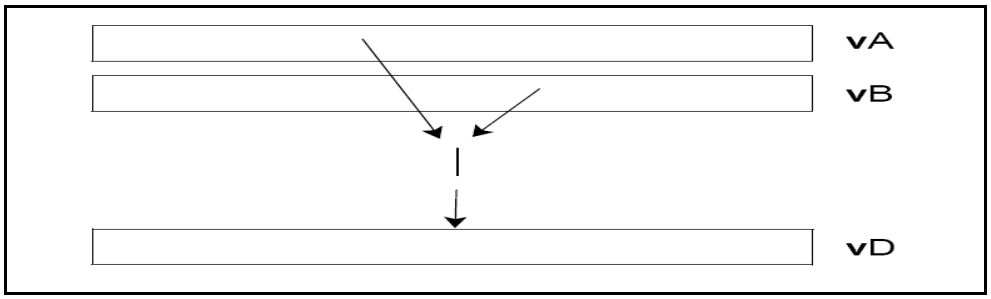

Figura 4.30 Instrucción VOR [2]

a.4) De rotación y desplazamiento.

Las instrucciones que abarcamos en este apartado son:

1. Rotación hacia la izquierda
2. Desplazamiento hacia la izquierda y desplazamiento hacia la derecha
3. Desplazamiento hacia la izquierda de todo el registro
4. Desplazamiento doble hacia la izquierda por inmediato
5. Desplazamiento hacia la izquierda por octetos
6. Desplazamiento hacia la derecha por octetos

1. Rotación hacia la izquierda

ROTACIÓN HACIA LA IZQUIERDA
vrlb
vrlh
vrlw

- Realiza la rotación hacia la izquierda de cada elemento de vA el número de bits especificado.
- El resultado lo almacenamos en la posición correspondiente en vD.

2. Desplazamiento hacia la izquierda y desplazamiento hacia la derecha.

DESPLAZAMIENTO A LA IZQUIERDA	DESPLAZAMIENTO A LA DERECHA
vslb	vsrb
vslh	vsrh
vslw	vsrw
	vsrab
	vsrah
	vsraw

- Realiza el desplazamiento hacia la izquierda (derecha) de cada elemento de vA el número de bits especificado.
- Los bits que salen del vector se pierden. Los que quedan vacantes se suplen con ceros.
- El resultado lo almacenamos en la posición correspondiente en vD.

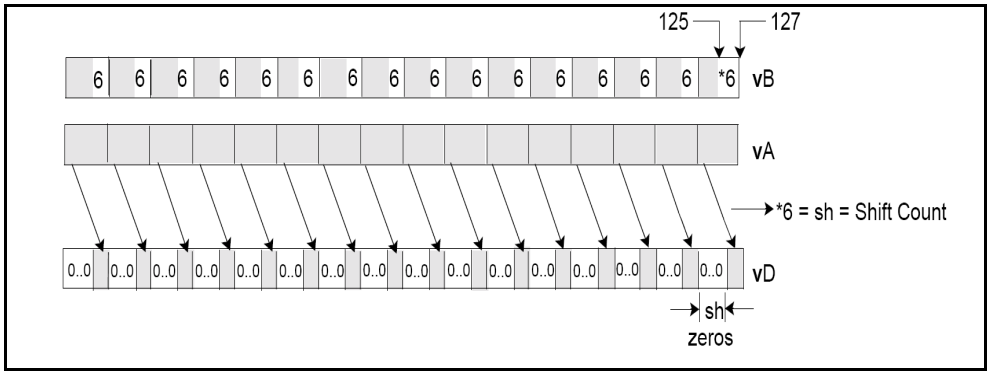

Figura 4.31 Instrucción VSRB [2]

3. Desplazamiento hacia la izquierda de todo el registro.

DESPLAZAMIENTO A LA IZQUIERDA
vsl

- El contenido de vA se desplaza a la izquierda el número de bits indicado por los 3 bits menos significativos de vB y el resultado se coloca en vD.

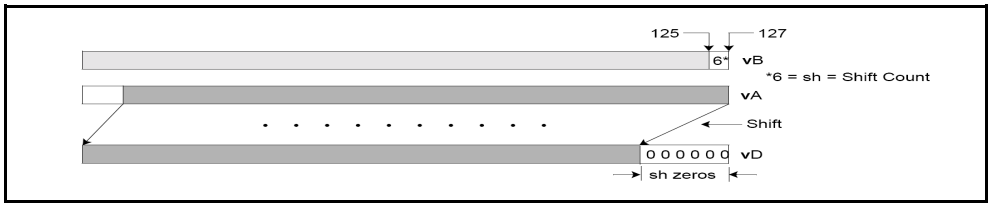

Figura 4.32 Instrucción VSL [2]

4. Desplazamiento doble a la izquierda por inmediato

DESPLAZAMIENTO DOBLE A LA IZQUIERDA POR INMEDIATO
vsldoi

- Coloca en vD el contenido de vA lo que indica SHB. Los que quedan vacantes se suplen por la parte alta de vB..

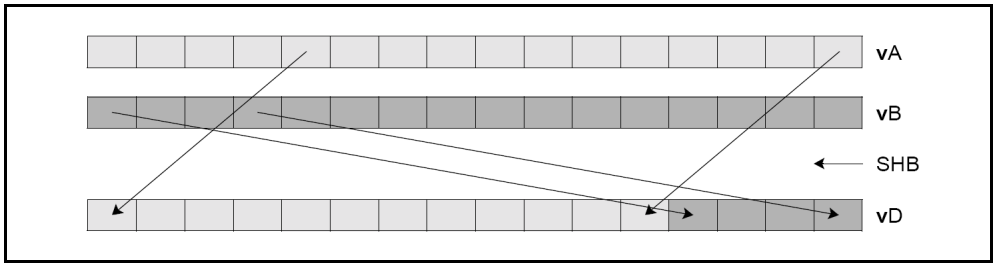

Figura 4.33 Instrucción VSLDOI [2]

5. Desplazamiento a la izquierda por octetos

DESPLAZAMIENTO A LA IZQUIERDA POR OCTETOS
vslo

- El contenido de vA se desplaza a la izquierda el número de bytes especificado por los bits [121-124] de vB.

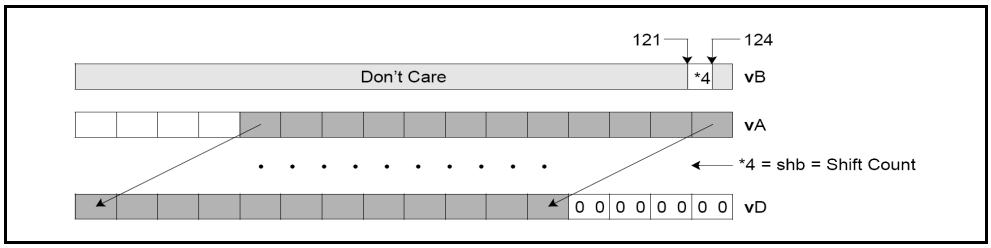

Figura 4.34 Instrucción VSLO [2]

6. Desplazamiento a la derecha por octetos

DESPLAZAMIENTO A LA DERECHA POR OCTETOS
vsro

- El contenido de vA se desplaza a la derecha el número de bytes especificado por los bits [121-124] de vB.

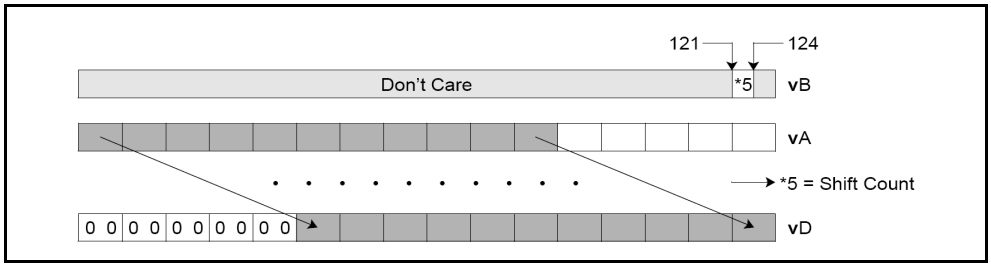

Figura 4.35 Instrucción VSRO [2]

Existen otro tipo de instrucciones vectoriales que no entrarían exactamente en la categoría de aritmética de enteros. Estos grupos son:

- *Instrucciones vectoriales de empaquetado y desempaquetado*
- *Instrucciones vectoriales de fusión*
- *Instrucciones vectoriales de "splat"*
- *Instrucción vectorial de permutación*
- *Instrucción vectorial de selección*

Instrucciones vectoriales de empaquetado y desempaquetado

Estas instrucciones se dividen en instrucciones vectoriales de empaquetado de:

1. Entero sin signo, módulo sin signo
2. Entero sin signo, saturación sin signo
3. Entero con signo, módulo sin signo
4. Entero con signo, módulo con signo
5. Pixel

y desempaquetado:

6. Parte alta, entero con signo
7. Parte alta, Pixel
8. Parte baja, entero con signo
9. Parte baja, Pixel

1. Empaquetado de entero sin signo, módulo sin signo.

ENTERO SIN SIGNO, MODULO SIN SIGNO
vpkuhum
vpkuwum

- Concatena los enteros sin signo de la parte baja de vA y los enteros sin signo de la parte baja de vB.
- El resultado lo coloca en vD usando módulo aritmético sin signo. Los elementos de vA se colocan en la parte baja de cada doble palabra de vD y cada elemento de vB se coloca en la parte alta.

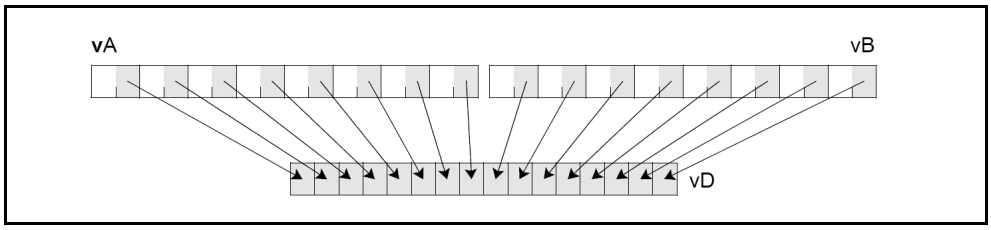

Figura 4.36 Instrucción VPKUHUM [2]

2. Empaquetado de entero sin signo, saturación sin signo

ENTERO SIN SIGNO, SATURACION SIN SIGNO
vpkuhus
vpkuwus

- Concatena los enteros sin signo de la parte baja de vA y de vB.
- El resultado lo coloca en vD usando saturación sin signo. Los elementos de vA se colocan en la parte baja de cada doble palabra de vD y cada elemento de vB se coloca en la parte alta.

3. Empaquetado de entero con signo,módulo sin signo

ENTERO CON SIGNO, MODULO SIN SIGNO
vpkshus
vpkswus

- Concatena los enteros con signo de la parte baja de vA y de vB.
- El resultado lo coloca en vD usando saturación sin signo. Los elementos de vA se colocan en la parte baja de cada doble palabra de vD y cada elemento de vB se coloca en la parte alta.

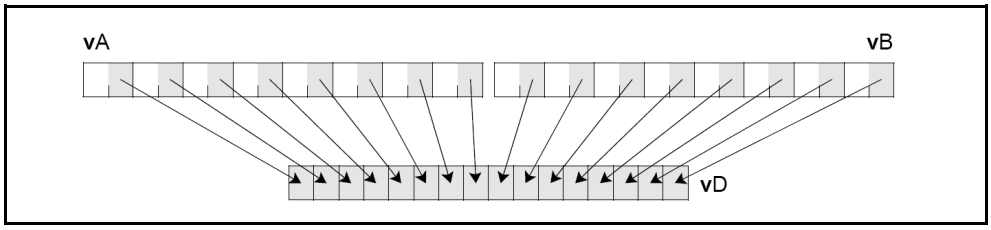

Figura 4.37 Instrucción VPKSHUS [2]

4. Empaquetado de entero con signo,módulo con signo

ENTERO CON SIGNO, MODULO CON SIGNO
vpkshss
vpkswss

- Concatena los enteros con signo de la parte baja de vA y de vB.
- El resultado lo coloca en vD usando saturación con signo. Los elementos de vA se colocan en la parte baja de cada doble palabra de vD y cada elemento de vB se coloca en la parte alta.

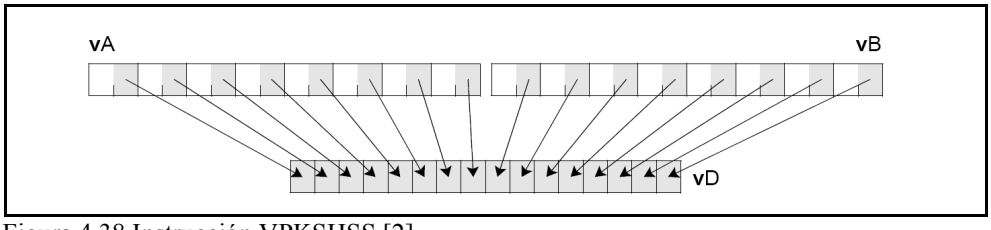

Figura 4.38 Instrucción VPKSHSS [2]

5. Pixel

PIXEL
vpkpx

- Cada palabra de vA y vB se empaqueta en media palabra. Cada media palabra se coloca en vD. Cada palabra se coloca según el siguiente orden:
  [bit 7 del primer byte (bit 7 de la palabra)]
  [bits 0-4 del segundo byte (bits 8-12 de la palabra)]
  [bits 0-4 del tercer byte (bits 16-20 de la palabra)]
  [bits 0-4 del cuarto byte (bits 24-28 de la palabra)]
- Cada media palabra de vA se coloca en la parte baja de la doble palabra de vD, y cada media palabra de vB se coloca en la parte alta de la doble palabra de vD.

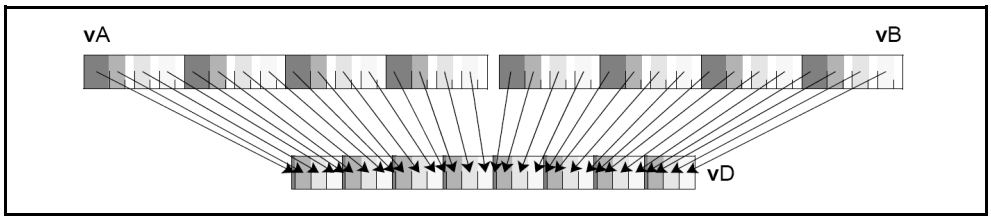

Figura 4.39 Instrucción VPKPX [2]

6. Desempaquetado parte alta, entero con signo

PARTE ALTA ENTERO CON SIGNO
vupkhsb
vupkhsh

- A cada elemento entero con signo de la parte alta de la doble palabra de vB se le extiende el signo y se coloca en vD.

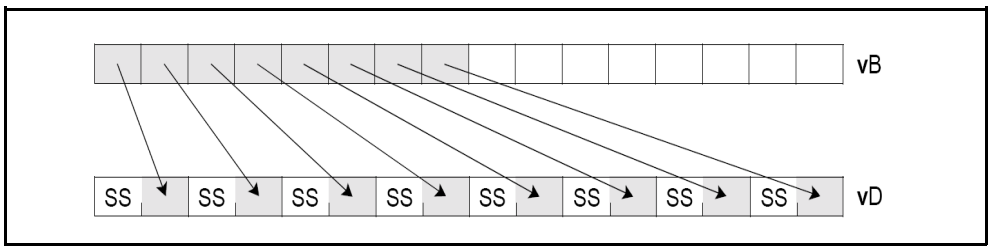

Figura 4.40 Instrucción VUPKHSB [2]

7. Desempaquetado parte alta, pixel

PARTE ALTA PIXEL
vupkhpx

- Cada media palabra de la parte alta de vB se desempaqueta para producir una palabra y colocarla en el mismo orden en vD.
- Cada media palabra se desempaqueta concatenando en orden el resultado de las siguientes operaciones:
  Extensión de signo del bit 0 de la media palabra a 8 bits
  Extensión del cero de los bits 1-5 de la media palabra a 8 bits
  Extensión del cero de los bits 6-10 de la media palabra a 8 bits
  Extensión del cero de los bits 11-15 de la media palabra a 8 bits

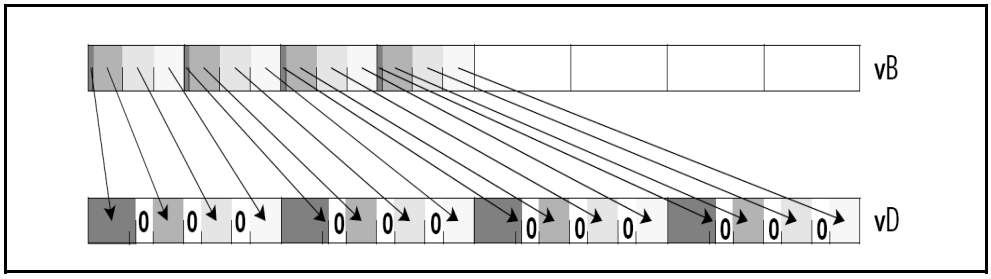

Figura 4.41 Instrucción VUPKHPX [2]

7. Desempaquetado parte baja, entero con signo

PARTE BAJA ENTERO CON SIGNO
vupklsb
vupklsh

- A cada elemento entero con signo de la parte baja de la doble palabra de vB se le extiende el signo y se coloca en vD.

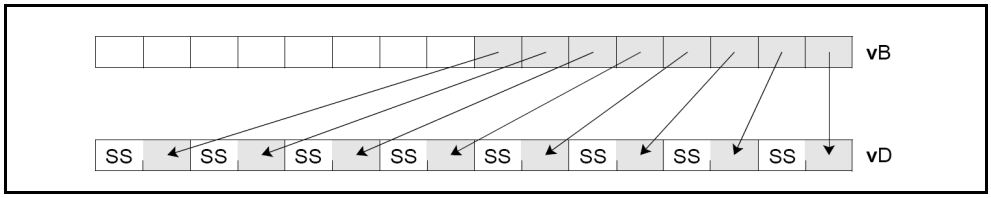

Figura 4.42 Instrucción VUPKLSB [2]

8. Desempaquetado parte baja, pixel

PARTE BAJA, PIXEL
vupklpx

- Cada media palabra de la parte baja de vB se desempaqueta para producir una palabra y colocarla en el mismo orden en vD.
- Cada media palabra se desempaqueta concatenando en orden el resultado de las siguientes operaciones:
  Extensión de signo del bit 0 de la media palabra a 8 bits
  Extensión del cero de los bits 1-5 de la media palabra a 8 bits
  Extensión del cero de los bits 6-10 de la media palabra a 8 bits
  Extensión del cero de los bits 11-15 de la media palabra a 8 bits

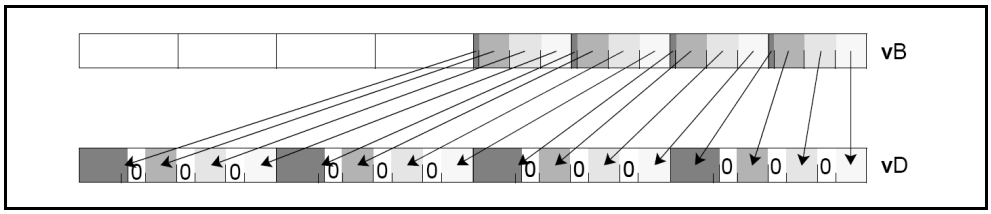

Figura 4.43 Instrucción VUPKLPX [2]

Instrucciones vectoriales de fusión

Las instrucciones que hacen referencia a la fusión o mezcla de enteros son:

1. Fusión de la parte entera
2. Fusión de la parte baja

1. Fusión de la parte entera

FUSION DE LA PARTE ENTERA
vmrghb
vmrghh
vmrghw

- Cada elemento entero en la parte alta de cada doble palabra en vA se coloca en la parte baja de cada elemento entero de vD. Cada elemento entero en la parte alta de cada doble palabra en vB se coloca en la parte alta de cada elemento entero de vD.

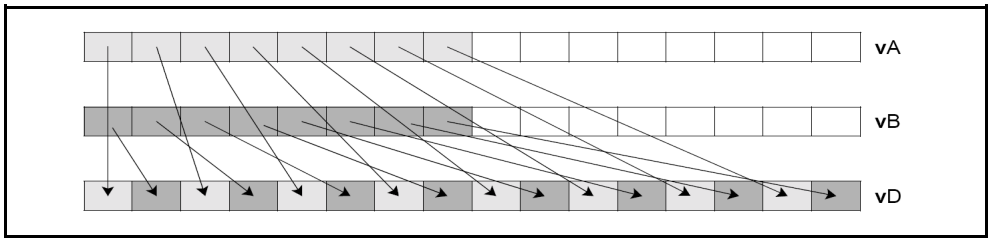

Figura 4.44 Instrucción VMRGHB [2]

2. Fusión de la parte baja

FUSION DE LA PARTE BAJA
vmrglb
vmrglh
vmrglw

- Cada elemento entero en la parte baja de cada doble palabra en vA se coloca en la parte baja de cada elemento entero de vD. Cada elemento entero en la parte baja de cada doble palabra en vB se coloca en la parte alta de cada elemento entero de vD.

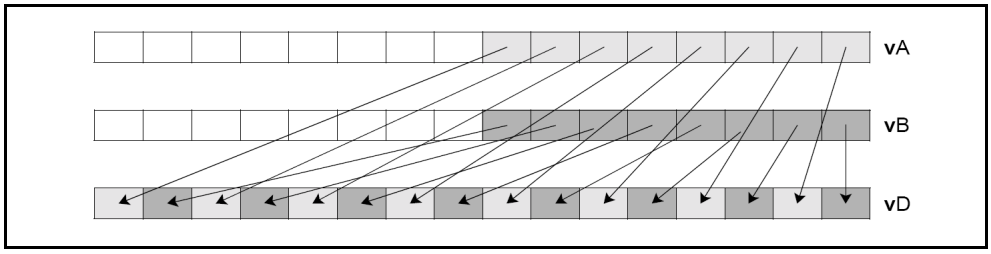

Figura 4.45 Instrucción VMRGLB [2]

Instrucciones vectoriales de "splat"

Este grupo de instrucciones vectoriales copian un mismo elemento en todas las subdivisiones del registro vectorial. Estas instrucciones son:

1. *Splat* entera
2. *Splat* entera inmediato con signo

1. *Splat* entera

SPLAT ENTERA
vspltb
vsplth
vspltw

- Copia el elemento de vB apuntado por el inmediato en cada una de las partes en las que esté dividido el registro vectorial vD.

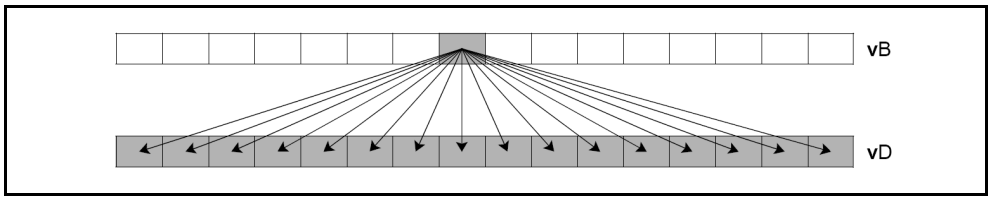

Figura 4.46 Instrucción VSPLTB [2]

2. *Splat* entera de inmediato con signo

SPLAT ENTERA INMEDIATO CON SIGNO
vspltisb
vspltish
vspltisw

- Se extiende el signo del inmediato y se coloca en vD.

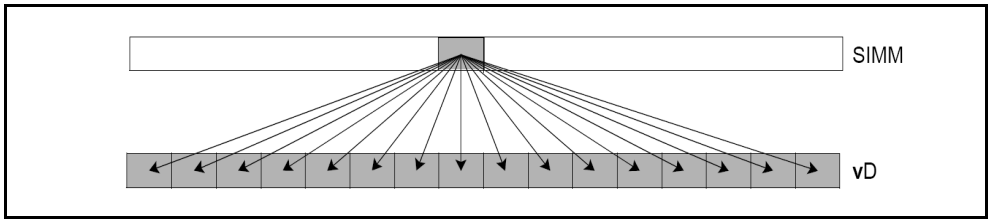

Figura 4.47 Instrucción VSPLTISB [2]

Instrucción vectorial de permutación

PERMUTACION VECTORIAL
vperm

- vC especifica que bytes de vA y vB se copian y colocan en cada byte de vD.

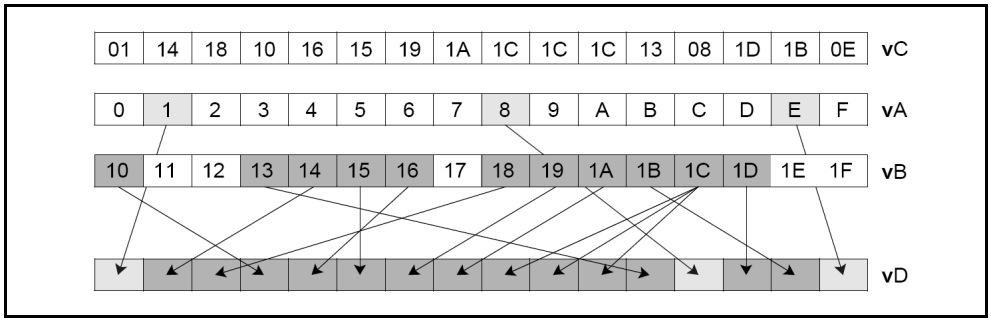

Figura 4.48 Instrucción VPERM [2]

Instrucción vectorial de selección

SELECCION VECTORIAL
vsel

- Cada bit de vC se compara con 0. Si es igual coloca el bit correspondiente de vA en vD, si no coloca el de vB.

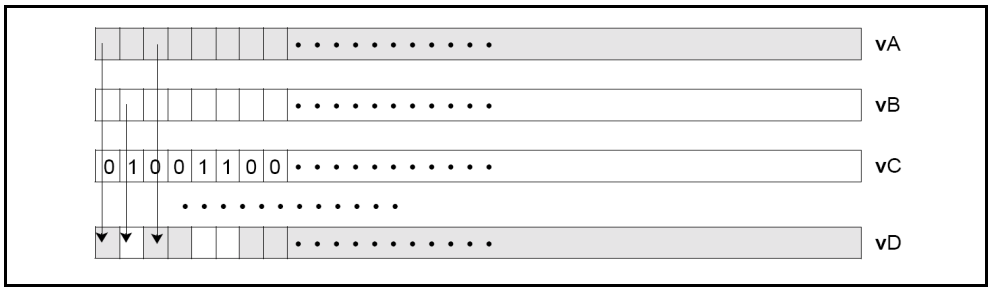

Figura 4.49 Instrucción VSEL [2]

b) Instrucciones vectoriales con punto flotante

Para la implementación de este tipo de instrucciones hay que tener en cuenta que se utilizan unos registros vectoriales especiales para punto flotante, los FPR.

Los métodos de redondeo se aplican internamente a los operandos en punto flotante.

La instrucciones que pertenecen a este grupo son:

1. Suma y resta
2. Máximo y mínimo
3. Multiplicación-suma y multiplicación-resta
4. De conversión y redondeo
5. De comparación
6. De estimación

1. Suma y resta

SUMA	RESTA
vaddfp	vsubfp

- Suma (Resta) cada elemento de vA en punto flotante al elemento correspondiente de vB.
- Redondea el resultado intermedio al número más cercado en precisión simple
- Almacenamos el resultado en la posición correspondiente de vD.

2. Máximo y mínimo

MAXIMO	MINIMO
vmaxfp	vminfp

- Compara cada elemento de vA en precisión simple con el correspondiente elemento de vB.
- El valor del máximo (mínimo) de los dos es almacenado en la posición correspondiente de vD.

3.Multiplicación-suma y multiplicación-resta

MULTIPLICACION SUMA	MULTIPLICACION RESTA
vmaddfp	vnmsubfp

- Mutiplica cada elemento de vA en punto flotante con el elemento correspondiente de vC.
- A este resultado le sumamos (restamos) el contenido del correspondiente elemento de vB.
- Redondea el resultado intermedio al número más cercado en precisión simple.
- Almacenamos el resultado en la posición correspondiente de vD.

4. De conversión y redondeo

REDONDEO	TIPO DE REDONDEO	CONVERSION	TIPO DE CONVERSION
vfin	Al más próximo	vcfux	De punto fijo sin signo
vrfiz	Hacia cero	vcfsx	De punto fijo con signo
vrfip	Hacia más infinito	vctuxs	A punto fijo sin signo
vrfin	Hacia menos infinito	vctsxs	A punto fijo con signo

5. De comparación

COMPARACION
vcmpgtfpx
vcmpeqfpx
vcmpgeqx
vcmpbfpx

- Compara cada elemento de vA en precisión simple con el elemento correspondiente de vB.
- Almacenamos el resultado de la comparación en la posición correspondiente de vD.

6. De estimación

ESTIMACION	OPERACIÓN
vrefp	Estimación
vrsqrtefp	Raiz cuadrada
vlogefp	Logaritmo en base 2
vexptefp	2 elevado al exponente

- Realiza la estimación de la operación indicada para cada elemento de vB.
- Almacena el resultado de las distintas operaciones en el lugar correspondiente de vD.

c) Instrucciones vectoriales de carga

Este tipo de instrucciones acceden a memoria y cargan el dato en vD. Para acceder a memoria se utiliza la dirección efectiva que se calcula previamente.

Las instrucciones vectoriales de carga son:

1. Carga Indexada
2. Carga y desplazamiento

1. Carga Indexada

CARGA INDEXADA
lvebx
lvehx
lvewx
lvx
lvxl

- Calcula la dirección efectiva sumando el contenido de rA y rB.
- Carga en vD el elemento de memoria situado en la dirección efectiva.

2. Carga y desplazamiento

CARGA Y DESPLAZAMIENTO
lvsl
lvsr

- Calcula la dirección efectiva sumando el contenido de rA y rB.
- Carga en vD un valor predefinido atendiendo al desplazamiento indicado por la dirección efectiva.

d) Instrucciones vectoriales de almacenamiento

Son instrucciones cuyo objetivo es el de almacenar un dato en memoria.

1. Almacenamiento indexado

ALMACENAMIENTO INDEXADO
svetbx
svethx
svetwx
stvx
stvxl

- Calcula la dirección efectiva sumando el contenido de rA y rB.
- Almacena el contenido de la parte baja de vS en la dirección de memoria indicada por la dirección efectiva.

4.2 Validación

Una vez implementadas las instrucciones AltiVec en nuestro simulador pasamos a la validación, es decir, dar respuesta a preguntas como: ¿qué optimiza?, ¿cuánto optimiza? y ¿en qué programas optimiza mejor?.

El algoritmo de Viterbi fue desarrollado por Andrew Viterbi como un esquema de corrección de errores para enlaces de comunicaciones digitales con ruido.

El algoritmo de Viterbi es un algoritmo de programación dinámico diseñado para encontrar el camino óptimo de los estados ocultos - conocidos como la trayectoria de Viterbi - como resultado de una secuencia de acontecimientos observados. El algoritmo de Viterbi es un algoritmo que está estrechamente relacionado con la computación de la probabilidad de una secuencia de acontecimientos observados. Es un algoritmo de búsqueda en anchura, capaz de encontrar una solución óptima en el espacio de búsqueda (de estados) generado.

El algoritmo realiza las siguientes tareas.

- Primero, tanto los sucesos observados como los escondidos deben estar en una secuencia. Esta secuencia se corresponde a menudo con el tiempo.
- Segundo, estas dos secuencias deben estar alineadas, y cada suceso observado debe corresponder exactamente con un suceso escondido.
- Y por último, debe computar el mejor camino posible de estados ocultos hacia un cierto punto *t*, que depende únicamente del suceso observado en el punto *t* y del mejor camino encontrado al punto *t-1*.

El algoritmo de Viterbi opera sobre una máquina de estados. Esto es, en cualquier momento el sistema que estamos modelando se encuentra en un determinado estado. Existen un número finito de estados almacenados en una lista. Cada estado representa un nodo. Múltiples secuencias de estados (caminos) pueden conducirte a un mismo estado, pero sólo uno es el camino óptimo (survivor path) para llegar a ese estado.

Aspectos fundamentales:

El algoritmo examinará todos los posibles caminos que conducen a un estado y sólo contemplará el mejor camino. De esta forma, el algoritmo no tiene que comprobar todos los caminos, sólo uno por estado.

Una transición de un estado previo a un nuevo estado es marcado por una métrica incremental, normalmente un número. Esta transición es computada desde el suceso.

Los sucesos se acumulan sobre el camino de alguna manera, normalmente sumados. Así que la base del algoritmo es almacenar un número para cada estado.

Cuando ocurre un nuevo suceso, el algoritmo camina moviéndose hacia un nuevo conjunto de estados combinando la métrica de un posible estado previo con la métrica incremental producida por la transición a este nuevo estado, y elige la mejor. La métrica incremental asociada con cada suceso depende de la posibilididad de que exista dicha transición del estado viejo al estado nuevo.

Los términos “trayectoria de Viterbi” y “algoritmo de Viterbi” también están relacionados con algoritmos de programación dinámica que se basan en encontrar la única explicación posible para una determinada observación.

En nuestro caso la utilización de este algoritmo nos permite conocer los errores que cometemos en el desarrollo de las instrucciones AltiVec. El cálculo del error acumulado es la única parte que puede ser realizada en paralelo.

En cuanto al tipo de errores más comunes que hemos ido cometiendo a lo largo del desarrollo del proyecto. Los comentamos a continuación

- Uno de los primeros problemas que tuvimos con la implementación fue la realización de *cast* sobre los datos que obteníamos de los registros, con lo que realizábamos operaciones sobre datos con signo cuando nuestra intención era operar con elementos sin signo. La solución de este contratiempo fue sencilla y rápida, ya que solo fue necesario observar el resultado almacenado en los registros destino.

- Los accesos a memoria nos causaron algún problema. Al principio decidimos utilizar accesos a memoria de tamaño *quadword*, lo cual nos ocasiono problemas de alineación de operandos, con lo que decidimos sustituir un acceso *quadword* por dos accesos de tamaño *word*, que aunque a primera vista parecía ineficiente, los resultados obtenidos fueron correctos en comparación los anteriores.

- Otro error común fue la duplicación de los códigos de operación. Las instrucciones AltiVec constan de un código de operación y otro extendido. En algunas de estas operaciones el código coincide con el código de un grupo de instrucciones genéricas del PowerPC, cosas que no comprobamos y por consiguiente se producían errores de duplicidad de código.

- Para solucionar este problema tuvimos que repasar el repertorio de instrucciones genéricas del PowerPC, y agrupar nuestras nuevas instrucciones con las ya existentes con dicho código de operación, diferenciándose entre si por el código de operación extendido.

- Por último el error más importante que tuvimos fue la implementación errónea de alguna instrucción, pero se resolvió fácilmente con la ayuda del depurador, para lo cual tuvimos que compilar nuestro simulador con la opción del GCC que nos expandía las macros del código para poder así depurar el código mediante la herramienta DDD.

En los apartados sucesivos mostramos los resultados obtenidos a lo largo del proyecto, tanto los resultados negativos como los resultados positivos una vez depurado las nuevas instrucciones.

4.2.1 Viterbi-dynamic con errores

** Simulation Started **
sim: ** starting *fast* functional simulation **
noise voltage = 0.447214
metric table range 4 to -78
original data:
2c2345675500000000000000055aaffff

frame 1 decoded data:
18acae6b6f869a69a69a69a65aa3866b
Seed 0 Amplitude 100 units, Eb/N0 = 10 dB
Amplitude 100 units, Eb/N0 = 10 dB
Frame length = 128 bits, #frames = 1, bits decoded = 128
frame errors: 1 (1)
bit errors: 61 (0.476562)
unimplemented syscall at 0x0fea4a4c, the syscall number is 234

sim: ** simulation statistics **

sim_num_insn	2396268449	total number of instructions executed
sim_elapsed_time	373	total simulation time in seconds
sim_inst_rate	6424312.1957	simulation speed (in insts/sec)
ld_text_base	0x10000000	program text (code) segment base
ld_text_size	13172	program text (code) size in bytes
ld_data_base	0x10013374	program initialized data segment base
ld_data_size	804	program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base	0x7fffc000	program stack segment base (highest address in stack)
ld_stack_size	1328	program initial stack size
ld_prog_entry	0x10000710	program entry point (initial PC)
ld_environ_base	0x7fffc004	program environment base address address
mem.page_count	510	current number of pages allocated
mem.page_mem	2040k	current size of memory pages allocated
mem.max_page_count	514	maximum number of pages allocated
mem.max_page_mem	2056k	current size of memory pages allocated
mem.ptab_misses	133	total first level page table misses
mem.ptab_accesses	3314436112	total page table accesses
mem.ptab_miss_rate	0.0000	first level page table miss rate

4.2.2 Viterbi-static con errores

** Simulation Started **
sim: ** starting *fast* functional simulation **
noise voltage = 0.447214
metric table range 4 to -78
original data:
2c2345675500000000000000055aaffff

frame 1 decoded data:
18acae6b6f869a69a69a69a65aa3866b

Seed 0 Amplitude 100 units, Eb/N0 = 10 dB
Amplitude 100 units, Eb/N0 = 10 dB
Frame length = 128 bits, #frames = 1, bits decoded = 128
frame errors: 1 (1)
bit errors: 61 (0.476562)
unimplemented syscall at 0x100254a8, the syscall number is 234

sim: ** simulation statistics **

sim_num_insn	2396017663	total number of instructions executed
sim_elapsed_time	239	total simulation time in seconds
sim_inst_rate	10025178.5063	simulation speed (in insts/sec)
ld_text_base	0x10000000	program text (code) segment base
ld_text_size	609428	program text (code) size in bytes
ld_data_base	0x100a4c94	program initialized data segment base
ld_data_size	3956	program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base	0x7fffc000	program stack segment base (highest address in stack)
ld_stack_size	2256	program initial stack size
ld_prog_entry	0x10000120	program entry point (initial PC)
ld_environ_base	0x7fffc004	program environment base address address
mem.page_count	162	current number of pages allocated
mem.page_mem	648k	current size of memory pages allocated
mem.max_page_count	164	maximum number of pages allocated
mem.max_page_mem	656k	current size of memory pages allocated
mem.ptab_misses	3	total first level page table misses
mem.ptab_accesses	3314524922	total page table accesses
mem.ptab_miss_rate	0.0000	first level page table miss rate

4.2.3 Viterbi-dynamic con los errores depurados

** Simulation Started **
sim: ** starting *fast* functional simulation **
noise voltage = 0.447214
metric table range 4 to -78
original data:
2c2345675500000000000000055aaffff

Seed 0 Amplitude 100 units, Eb/N0 = 10 dB
Amplitude 100 units, Eb/N0 = 10 dB
Frame length = 128 bits, #frames = 1, bits decoded = 128
frame errors: 0 (0)
bit errors: 0 (0)
unimplemented syscall at 0x0fea4a4c, the syscall number is 234

sim: ** simulation statistics **

sim_num_insn	2214023662	total number of instructions executed
sim_elapsed_time	342	total simulation time in seconds
sim_inst_rate	6473753.3977	simulation speed (in insts/sec)
ld_text_base	0x10000000	program text (code) segment base
ld_text_size	13172	program text (code) size in bytes

ld_data_base	0x10013374	program initialized data segment base
ld_data_size	804	program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base	0x7fffc000	program stack segment base (highest address in stack)
ld_stack_size	1328	program initial stack size
ld_prog_entry	0x10000710	program entry point (initial PC)
ld_environ_base	0x7fffc004	program environment base address address
mem.page_count	510	current number of pages allocated
mem.page_mem	2040k	current size of memory pages allocated
mem.max_page_count	514	maximum number of pages allocated
mem.max_page_mem	2056k	current size of memory pages allocated
mem.ptab_misses	133	total first level page table misses
mem.ptab_accesses	3100595272	total page table accesses
mem.ptab_miss_rate	0.0000	first level page table miss rate

4.2.4 Viterbi-static con los errores depurados

** Simulation Started **
sim: ** starting *fast* functional simulation **
noise voltage = 0.447214
metric table range 4 to -78
original data:
2c2345675500000000000000055aaffff

Seed 0 Amplitude 100 units, Eb/N0 = 10 dB
Amplitude 100 units, Eb/N0 = 10 dB
Frame length = 128 bits, #frames = 1, bits decoded = 128
frame errors: 0 (0)
bit errors: 0 (0)
unimplemented syscall at 0x100254a8, the syscall number is 234
sim: ** simulation statistics **

sim_num_insn	2213772070	total number of instructions executed
sim_elapsed_time	336	total simulation time in seconds
sim_inst_rate	6588607.3512	simulation speed (in insts/sec)
ld_text_base	0x10000000	program text (code) segment base
ld_text_size	609428	program text (code) size in bytes
ld_data_base	0x100a4c94	program initialized data segment base
ld_data_size	3956	program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base	0x7fffc000	program stack segment base (highest address in stack)
ld_stack_size	1328	program initial stack size
ld_prog_entry	0x10000120	program entry point (initial PC)
ld_environ_base	0x7fffc004	program environment base address address
mem.page_count	162	current number of pages allocated
mem.page_mem	648k	current size of memory pages allocated
mem.max_page_count	164	maximum number of pages allocated
mem.max_page_mem	656k	current size of memory pages allocated
mem.ptab_misses	3	total first level page table misses
mem.ptab_accesses	3100682725	total page table accesses
mem.ptab_miss_rate	0.0000	first level page table miss rate

4.2.5 Viterbi-Dynamic con todas las instrucciones implementadas

** Simulation Started **
sim: ** starting *fast* functional simulation **
noise voltage = 0.447214
metric table range 4 to -78
original data:
2c2345675500000000000000055aaffff

Seed 0 Amplitude 100 units, Eb/N0 = 10 dB
Amplitude 100 units, Eb/N0 = 10 dB
Frame length = 128 bits, #frames = 1, bits decoded = 128
frame errors: 0 (0)
bit errors: 0 (0)
unimplemented syscall at 0x0fea4a4c, the syscall number is 234

sim: ** simulation statistics **

sim_num_insn	2214023681	total number of instructions executed
sim_elapsed_time	367	total simulation time in seconds
sim_inst_rate	3565255.5250	simulation speed (in insts/sec)
ld_text_base	0x10000000	program text (code) segment base
ld_text_size	13172	program text (code) size in bytes
ld_data_base	0x10013374	program initialized data segment base
ld_data_size	804	program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base	0x7fffc000	program stack segment base (highest address in stack)
ld_stack_size	1296	program initial stack size
ld_prog_entry	0x10000710	program entry point (initial PC)
ld_environ_base	0x7fffc004	program environment base address address
mem.page_count	510	current number of pages allocated
mem.page_mem	2040k	current size of memory pages allocated
mem.max_page_count	514	maximum number of pages allocated
mem.max_page_mem	2056k	current size of memory pages allocated
mem.ptab_misses	133	total first level page table misses
mem.ptab_accesses	3100595190	total page table accesses
mem.ptab_miss_rate	0.0000	first level page table miss rate

4.2.6 Viterbi-Static con todas las instrucciones implementadas

** Simulation Started **
sim: ** starting *fast* functional simulation **
noise voltage = 0.447214
metric table range 4 to -78
original data:
2c2345675500000000000000055aaffff
Seed 0 Amplitude 100 units, Eb/N0 = 10 dB
Amplitude 100 units, Eb/N0 = 10 dB
Frame length = 128 bits, #frames = 1, bits decoded = 128
frame errors: 0 (0)
bit errors: 0 (0)

unimplemented syscall at 0x100254a8, the syscall number is 234

sim: ** simulation statistics **

sim_num_insn	2213771816	total number of instructions executed
sim_elapsed_time	370	total simulation time in seconds
sim_inst_rate	3553405.8042	simulation speed (in insts/sec)
ld_text_base	0x10000000	program text (code) segment base
ld_text_size	609428	program text (code) size in bytes
ld_data_base	0x100a4c94	program initialized data segment base
ld_data_size	3956	program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base	0x7fffc000	program stack segment base (highest address in stack)
ld_stack_size	1292	program initial stack size
ld_prog_entry	0x10000120	program entry point (initial PC)
ld_environ_base	0x7fffc004	program environment base address address
mem.page_count	162	current number of pages allocated
mem.page_mem	648k	current size of memory pages allocated
mem.max_page_count	164	maximum number of pages allocated
mem.max_page_mem	656k	current size of memory pages allocated
mem.ptab_misses	3	total first level page table misses
mem.ptab_accesses	3100682347	total page table accesses
mem.ptab_miss_rate	0.0000	first level page table miss rate

4.2.7 Viterbi-Sin-AltiVec

**** Simulation Started ****
sim: ** starting *fast* functional simulation **
noise voltage = 0.447214
metric table range 4 to -78
original data:
2c23456755000000000000055aaffff

Seed 267724512 Amplitude 100 units, Eb/N0 = 10 dB
Amplitude 100 units, Eb/N0 = 10 dB
Frame length = 128 bits, #frames = 1, bits decoded = 128
frame errors: 0 (0)
bit errors: 0 (0)
unimplemented syscall at 0x0fea525c, the syscall number is 234

sim: ** simulation statistics **

sim_num_insn	1824191	total number of instructions executed
sim_elapsed_time	1	total simulation time in seconds
sim_inst_rate	1824191.0000	simulation speed (in insts/sec)
ld_text_base	0x10000000	program text (code) segment base
ld_text_size	18092	program text (code) size in bytes
ld_data_base	0x100146ac	program initialized data segment base
ld_data_size	804	program init'ed `.data' and uninit'ed `.bss' size in bytes

ld_stack_base	0x7fffc000	program stack segment base (highest address in stack)
ld_stack_size	1344	program initial stack size
ld_prog_entry	0x10000610	program entry point (initial PC)
ld_environ_base	0x7fffc004	program environment base address address
mem.page_count	511	current number of pages allocated
mem.page_mem	2044k	current size of memory pages allocated
mem.max_page_count	515	maximum number of pages allocated
mem.max_page_mem	2060k	current size of memory pages allocated
mem.ptab_misses	138	total first level page table misses
mem.ptab_accesses	2873222	total page table accesses
mem.ptab_miss_rate	0.0000	first level page table miss rate

5 CONCLUSIÓN

La importancia de los simuladores arquitectónicos reside en la capacidad de realizar pruebas sobre diversas arquitecturas y modificar diversos aspectos de la misma para así obtener los resultados que más se acerquen a los deseados.

Es interesante para realizar simulaciones de procesadores que todavía no existen o que no se dispone de ellos físicamente y así obtener los resultados de ganancia, velocidad, etc, de una forma fiable y más barata de lo que costaría realizar un prototipo. Nos permite verificar el correcto funcionamiento de los programas así como medir su rendimiento sin necesidad de trabajar directamente sobre la arquitectura sobre la que han sido compilados.

Es importante remarcar la posibilidad de modificar diversos aspectos de la arquitectura para poder mejorar la ganancia y la velocidad de proceso sin la necesidad de tener el hardware específico, pero obteniendo resultados a todas luces orientativos del camino a seguir a la hora de diseñar dichas arquitecturas.

Este proyecto nos ha dado la posibilidad de conocer el funcionamiento de una de estas herramienta de simulación: el SimpleScalar. Hemos podido comprobar el comportamiento del proceso vectorial sin la necesidad de tener a mano una arquitectura específica, pudiendo comprobar el resultado de operaciones en paralelo así como su eficiencia.

Para trabajo futuro sobre el presente proyecto queda el campo de las comprobaciones con programas reales, tales como MiBench o EEMBC.

También sería interesante trasladar el proyecto a un simulador más complejo como pueda ser el sim-outorder, para así poder modificar las latencias de las instrucciones y obtener los resultados de las diversas ejecuciones, y de esta constatar la importancia a la hora de diseñar un nuevo hardware o modificar uno ya existente.

Por último, cabe la posibilidad de añadir más instrucciones al repertorio de instrucciones AltiVec, de las ya implementadas, y modificar el compilador GCC (sólo la parte de generación de código).

6 BIBLIOGRAFÍA

[1] IBM, 2003, *"PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-Bit Microprocessors"*, Versión 2.0

[2] IBM,2003,*"PowerPC Microprocessor Family: AltiVec™* Technology Programming Environments Manual for 64 and 32-Bit Microprocessors"*, Versión 2.0

[3] Karthikeyan Sankaralingam,2004,"SimpleScalar Simulation of the PowerPC Instruction Set Architecture"

[4] Wesley F. Miaw, 1999, "*The Motorola AltiVec™ Technology*"

[5] Liberty, Jesse,2003, Programming C#,O'ReillyLiberty, Jesse,2003

[6] http://www03.ibm.com/chips/power/PowerPC/

[7] http://www.freescale.com/

[8] http://www.simdtech.org/AltiVec

[9] http://www.freescale.com/files/32bit/doc/fact_sheet/AltiVecFACT.pdf

[10] http://developer.apple.com/hardwaredrivers/ve/summary.html

[11] http://www.simplescalar.com/

[12] http://www-ali.cs.umass.edu/DSS/

[13] http://en.wikipedia.org/wiki/AltiVec

[14] http://www-numi.fnal.gov/offline_software/external_pkgs/Linux2.4-GCC_3_4/valgrind-3.1.0/none/tests/ppc32/jm-vmx.stdout.exp

[15] http://www.lightsoft.co.uk/Fantasm/avec_prog.html

[16] http://sourceware.org/ml/binutils/2000-05/msg00035.html

[17] http://cvs.sf.net/viewcvs.py/pearpc/pearpc/src/cpu/cpu_generic/ppc_vec.cc?
http://cvs.sf.net/viewcvs.py/pearpc/pearpc/src/cpu/cpu_generic/ppc_vec.cc
rev=1.3

[18] http://www.zator.com/Cpp/E2_2_4a.htm

[19] http://www.mactech.com/articles/mactech/Vol.15/15.07/AltiVecRevealed/

[20] http://www.fdi.ucm.es

AUTORIZACIÓN

Por la presente autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria como el código, la documentación y/o el prototipo desarrollado.

Jacobo Cadavid Zaragoza	**Jon Crespo Anasagasti**	**José Manuel Díaz Ruíz**
07243975-X	**02652328-Z**	**01937064-G**

En Madrid, a de Julio de 2006.